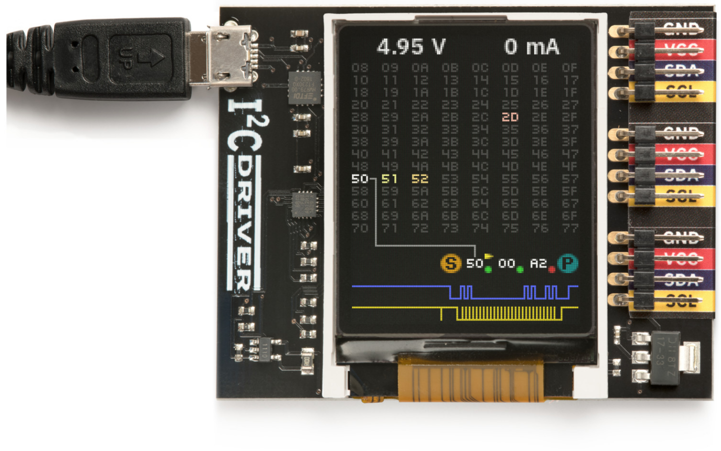


Last updated on January 28, 2020

Contents

1 Overview

I^{2}CDriver is an easy-to-use, open source tool for controlling I^{2}C devices. It works with Windows, Mac, and Linux, and has a built-in color screen that shows a live dashboard of all the I^{2}C activity. It uses a standard FTDI USB serial chip to talk to the PC, so no special drivers need to be installed. The board includes a separate 3.3 V supply with voltage and current monitoring.

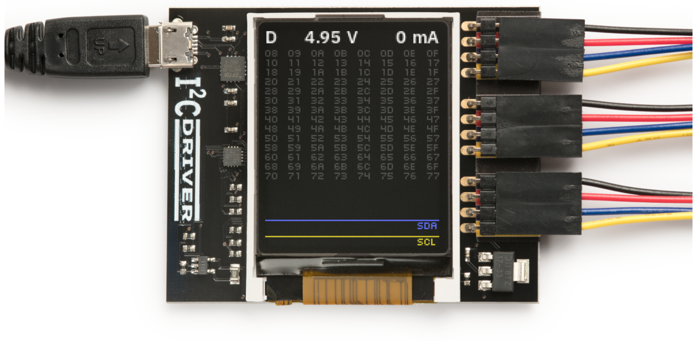
I^{2}CMini is a reduced-size version of I^{2}CDriver suitable for use in embedded devices. It is 100% compatible with I^{2}CDriver but lacks a display, current or voltage monitoring. In every other respect it behaves identically to I^{2}CDriver.

1.1 Features

- **Live display**: shows you exactly what it's doing all the time
- **Supports all I^{2}C features**: 7- and 10-bit I^{2}C addressing, clock stretching, bus arbitration, and sustained I^{2}C transfers at 400 and 100 kHz
- **I^{2}C pullups**: programmable I^{2}C pullup resistors, with automatic tuning
- **USB voltage monitoring**: USB line voltage monitor to detect supply problems, to 0.01 V
- **Target power monitoring**: target device high-side current measurement, to 5 mA
- **Three I^{2}C ports**: three identical I^{2}C ports, each with power and I^{2}C signals
- **Jumpers**: three sets of high-quality color coded 100mm jumpers included
- **3.3 V output**: output levels are 3.3 V, all are 5 V tolerant
- **Sturdy componentry**: uses an FTDI USB serial adapter, and Silicon Labs automotive-grade EFM8 controller
- **Open hardware**: the design, firmware and all tools are under BSD license
- **Flexible control**: GUI, command-line, C/C++, and Python 2/3 host software provided for Windows, Mac, and Linux

2 Getting Started

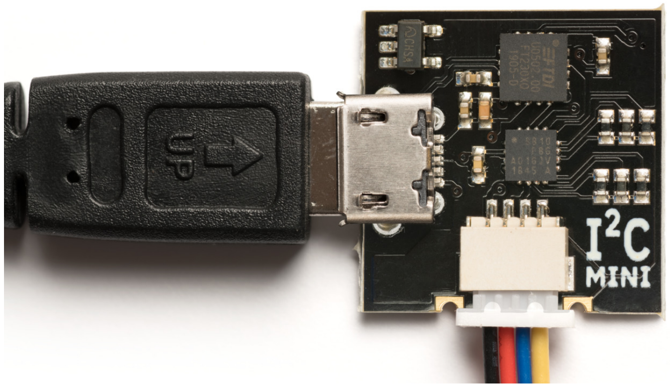
When you first connect I²CDriver to the USB port, the display blinks white for a moment then displays the initial status screen:



Connect the three sets of colored hookup wires as shown, following the same sequence as on the colored label:

GND	black	signal ground
VCC	red	3.3 V supply
SDA	blue	I²C data
SCL	yellow	I²C clock

Across the top of the display I²CDriver continuously shows the USB bus voltage and the current output.



I^2CMini also has a micro USB connector. Its Qwiic connector is polarized; it attaches as shown. The color coding for the signals is the same as for I^2CDriver.

3 Software installation

The source for all the I^{2}CDriver software is the repository. Available are:

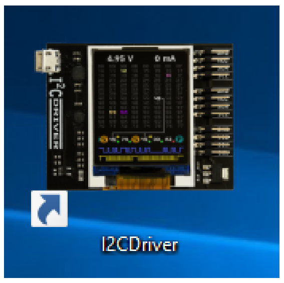
- a Windows/Mac/Linux GUI
- a Windows/Mac/Linux command-line
- Python 2 and 3 bindings
- Windows/Mac/Linux C/C++ bindings

Installation of the GUI and command-line utilities varies by platform.

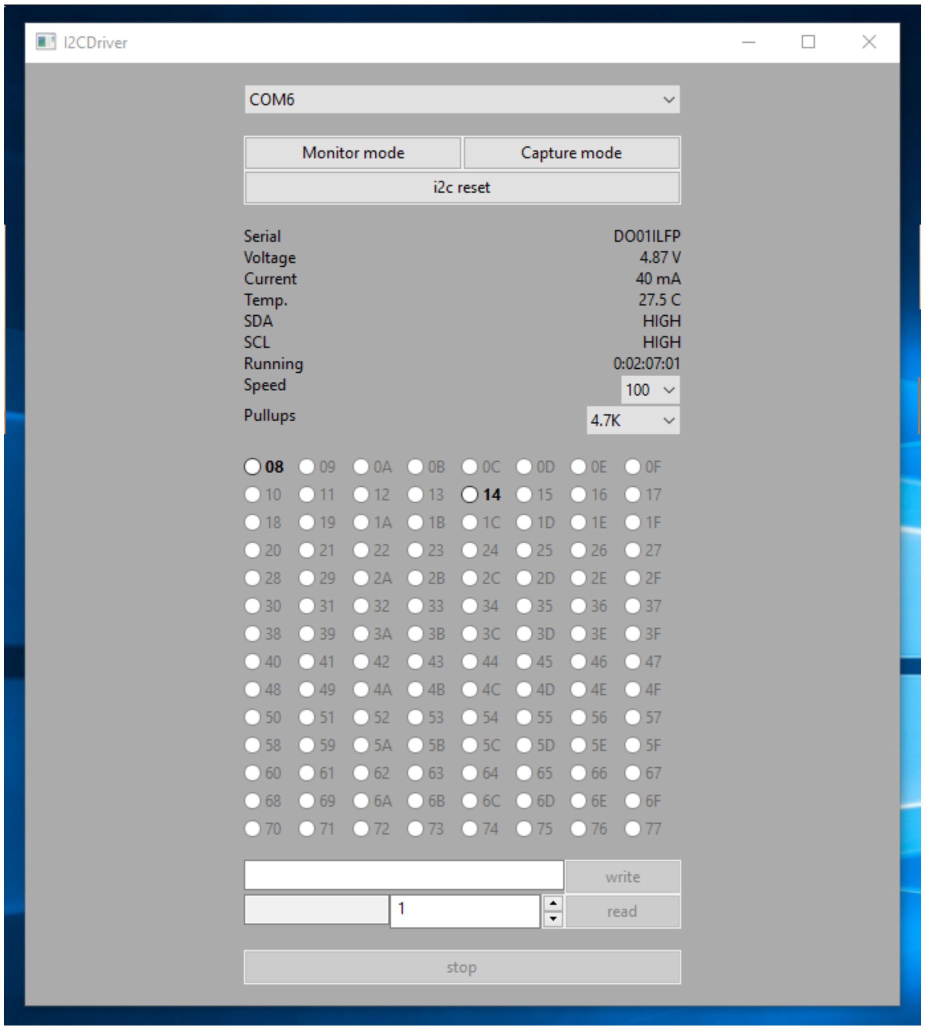
3.1 Windows

This installer contains the GUI and command-line utilities.

The GUI shortcut is installed on the desktop:



launching it brings up the control window:



Serial	DO01ILFP
Voltage	4.87 V
Current	40 mA
Temp.	27.5 C
SDA	HIGH
SCL	HIGH
Running	0:02:07:01
Speed	100
Pullups	4.7K

If there is only one serial device, the I^{2}CDriver device should be automatically selected. If there is more than one device, select its COM port from the pull-down menu at the top. Once connected, you can select a connected I^{2}C device and write and read data.

The command line utility `i2ccl` is also installed, and added to the Windows `PATH`. For example to display status information:

```
c:\>i2ccl COM6 i
uptime 8991 4.957 V 30 mA 25.8 C SDA=1 SCL=1 speed=100 kHz
```

See below for more information on the command-line syntax.

3.2 Linux

The GUI is included in the `i2cdriver` Python package, compatible with both Python 2 and 3. To install it, open a shell prompt and do:

```
sudo pip install i2cdriver
```

Then run it with

```
i2cgui.py
```

For the command-line tool, clone the repository, then do:

```
cd i2cdriver/c
make -f linux/Makefile
sudo make -f linux/Makefile install
i2ccl /dev/ttyUSB0 i
```

and you should see something like:

```
uptime 1651 4.971 V 0 mA 21.2 C SDA=1 SCL=1 speed=100 kHz
```

3.3 MacOS

The GUI is included in the `i2cdriver` Python package, compatible with both Python 2 and 3. To install it, open a shell prompt and do:

```
sudo pip install i2cdriver
```

Then run it with

```
i2cgui.py
```

For the command-line tool, clone the repository , then do:

```
cd i2cdriver/c
make -f linux/Makefile
sudo make -f linux/Makefile install
i2ccl /dev/cu.usbserial-DO00QS8D i
```

(substituting your actual I^{2}CDriver's ID for `DO00QS8D`) and you should see something like:

```
uptime 1651 4.971 V 5 mA 21.2 C SDA=1 SCL=1 speed=100 kHz
```

Note that the port to use is `/dev/cu.usbserial-XXXXXXXX`, as explained here.

4 APIs

4.1 Python 2 and 3

The I^{2}CDriver bindings can be installed with `pip` like this:

```
pip install i2cdriver
```

then from Python you can read an LM75B temperature sensor with:

```
>>> import i2cdriver
>>> i2c = i2cdriver.I2CDriver("/dev/ttyUSB0")
>>> d=i2cdriver.EDS.Temp(i2c)
>>> d.read()
17.875
>>> d.read()
18.0
```

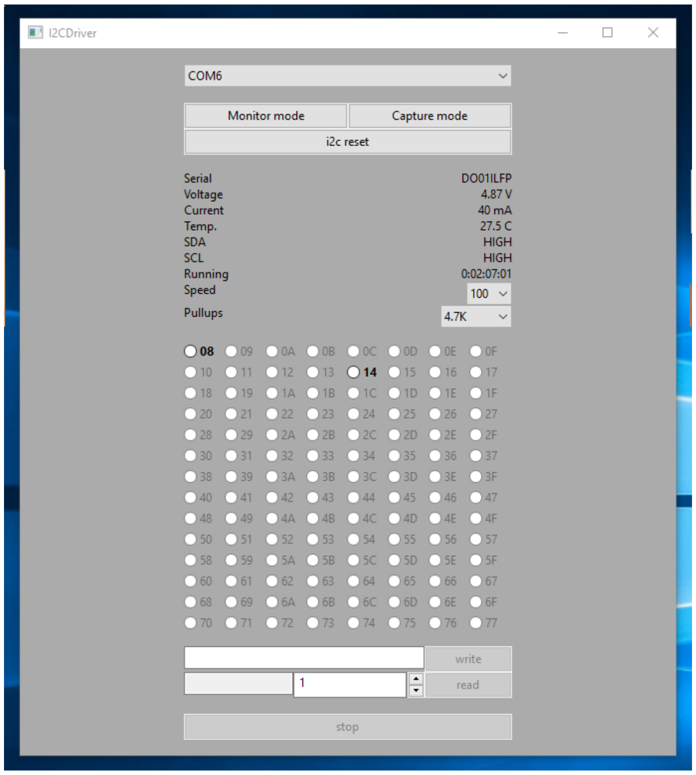
You can print a bus scan with:

```
>>> i2c.scan()
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- 1C -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
48 -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
68 -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
[28, 72, 104]
```

The Python GUI (which uses wxPython) can be run with:

```
python i2cgui.py
```

which depending on your distribution looks something like this:



```
Serial       DO01ILFP
Voltage      4.87 V
Current      40 mA
Temp.        27.5 C
SDA          HIGH
SCL          HIGH
Running      0:02:07:01
Speed        100
Pullups      4.7K
```

There are more examples in the samples folder in the repository.

The module has extensive help strings:

```
>>> help(i2cdriver)
```

displays the API documentation.

4.1.1 Reference

`class i2cdriver.I2CDriver(`*port='/dev/ttyUSB0', reset=True*`)`

A connected I2CDriver.

Variables

- `product` – product code e.g. i2cdriver1
- `serial` – serial string of I2CDriver
- `uptime` – time since I2CDriver boot, in seconds
- `voltage` – USB voltage, in V
- `current` – current used by attached device, in mA
- `temp` – temperature, in degrees C
- `scl` – state of SCL
- `sda` – state of SDA
- `speed` – current device speed in KHz (100 or 400)
- `mode` – IO mode (I2C or bitbang)
- `pullups` – programmable pullup enable pins
- `ccitt_crc` – CCITT-16 CRC of all transmitted and received bytes

`__init__(`*port='/dev/ttyUSB0', reset=True*`)`

Connect to a hardware i2cdriver.

Parameters

- `port` (*str*) – The USB port to connect to
- `reset` (*bool*) – Issue an I2C bus reset on connection

`setspeed(`*s*`)`

Set the I2C bus speed.

Parameters `s` (*int*) – speed in KHz, either 100 or 400

`setpullups(`*s*`)`

Set the I2CDriver pullup resistors

Parameters `s` – 6-bit pullup mask

`scan`(*silent=False*)

Performs an I2C bus scan. If silent is False, prints a map of devices. Returns a list of the device addresses.

```
>>> i2c.scan()
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- 1C -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
48 -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
68 -- -- -- -- -- -- --
-- -- -- -- -- -- -- --
[28, 72, 104]
```

`reset`()

Send an I2C bus reset

`start`(*dev*, *rw*)

Start an I2C transaction

Parameters

- `dev` – 7-bit I2C device address
- `rw` – read (1) or write (0)

To write bytes `[0x12,0x34]` to device `0x75`:

```
>>> i2c.start(0x75, 0)
>>> i2c.write([0x12,034])
```

(continues on next page)

(continued from previous page)

```
>>> i2c.stop()
```

`read`(*l*)

Read l bytes from the I2C device, and NAK the last byte

`write`(*bb*)

Write bytes to the selected I2C device

Parameters `bb` – sequence to write

`stop`()

stop the i2c transaction

`regrd`(*dev*, *reg*, *fmt='B'*)

Read a register from a device.

Parameters

- `dev` – 7-bit I2C device address
- `reg` – register address 0-255
- `fmt` – `struct.unpack()` format string for the register contents

If device 0x75 has a 16-bit register 102, it can be read with:

```
>>> i2c.regrd(0x75, 102, ">H")
4999
```

`regwr`(*dev*, *reg*, **vv*)

Write a devices register.

Parameters

- `dev` – 7-bit I2C device address
- `reg` – register address 0-255
- `vv` – sequence of values to write

To set device 0x34 byte register 7 to 0xA1:

```
>>> i2c.regwr(0x34, 7, [0xa1])
```

If device 0x75 has a big-endian 16-bit register 102 you can set it to 4999 with:

```
>>> i2c.regwr(0x75, 102, struct.pack(">H", 4999))
```

`monitor(s)`

Enter or leave monitor mode

Parameters `s` – `True` to enter monitor mode, `False` to leave

`getstatus()`

Update all status variables

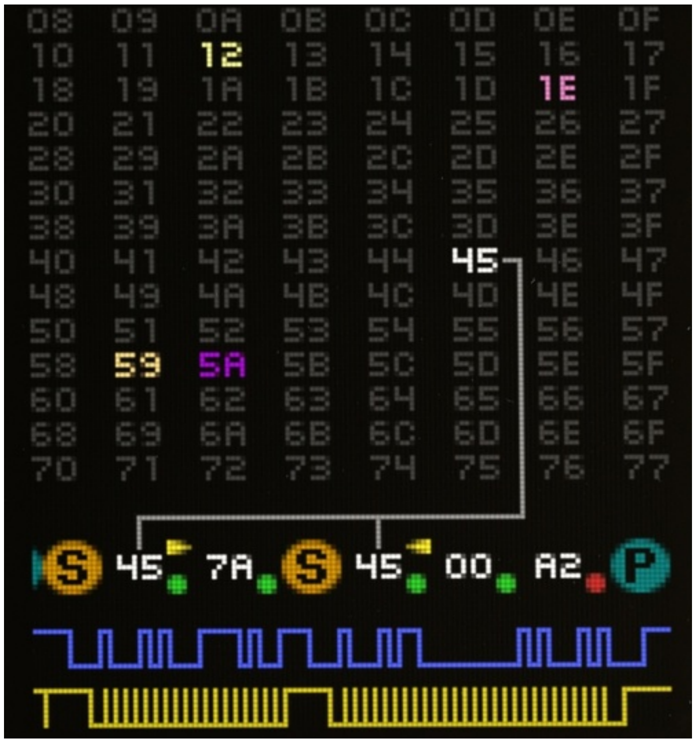
4.2 C/C++

I²CDriver is contained in a single source file with a single header. Both are in this subdirectory. Usage follows the Python API and is fairly self-explanatory.

5 Using I²CDriver

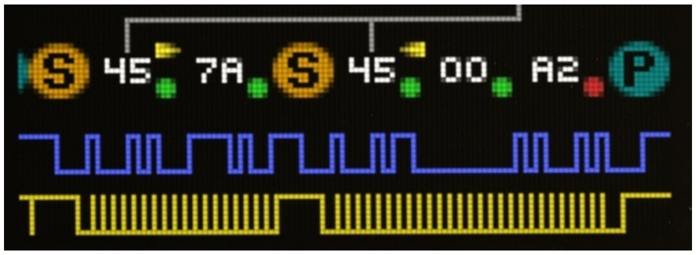
5.1 The display

The main display on the screen has three sections. The top section is a heat-map showing all 112 legal I²C addresses. Addresses that are currently active are white. Inactive addresses fade to yellow, purple and finally blue. The middle section is a symbolic interpretation of current I²C traffic. Details on this are below. The bottom two lines show a representation of the SDA (blue) and SCL (yellow) signals.



The symbolic decode section shows I²C transactions as they happen. Start and stop are shown as S and P symbols. After a S symbol the address byte is shown, with a right arrow (write) or left arrow (read). The gray lines connect the address byte to its heat-map indicator. Following this is a series of data bytes. Each byte is shown in hex, with either a green dot (ACK) or red dot

(NACK).



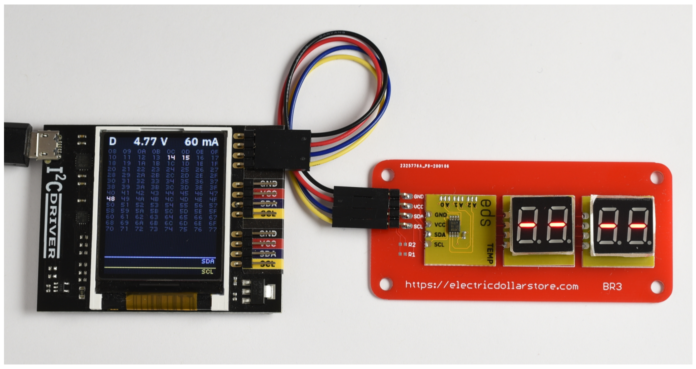
So for example the above sequence is showing

- Start, write to address `45`
- Write byte `7A`
- Repeated Start, read from address `45`
- Read byte `00`
- Read byte `A2`
- Stop

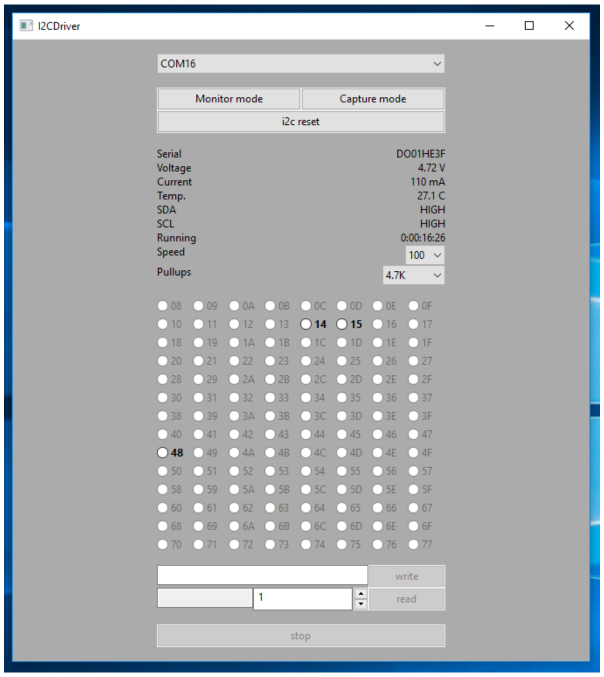
The above sequence is very typical for reading registers from an I^{2}C Device. Note that the final NACK (red dot) is not an error condition, but the standard way of handling the last byte of read transaction.

5.2 The GUI

The GUI is a straightforward way of interacting with I²C devices. For example, here I²CDriver is connected to some I²C peripherals: an LM75B temperature sensor and two 7-segment display modules.



Starting the GUI and connecting to the I²CDriver on `COM16` shows the temperature sensor at address `48` and the two display modules at addresses `14` and `15`.

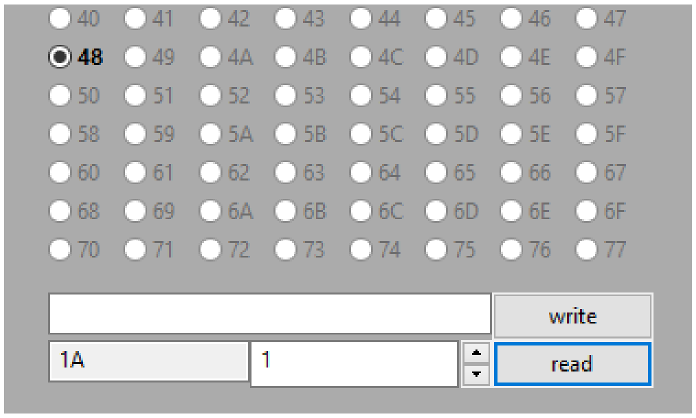


COM16

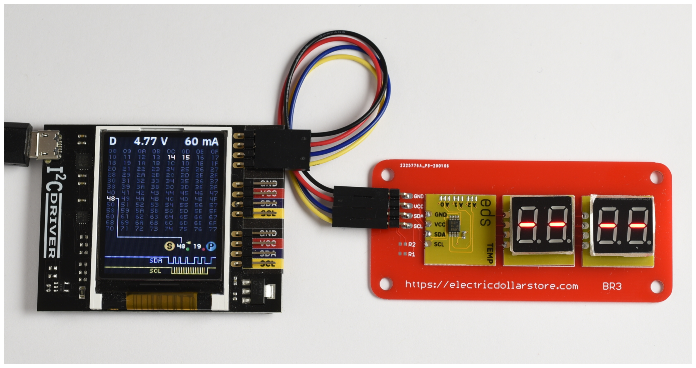
Serial	DO01HE3F
Voltage	4.72 V
Current	110 mA
Temp.	27.1 C
SDA	HIGH
SCL	HIGH
Running	0:00:16:26
Speed	100
Pullups	4.7K

08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	**14**	**15**	16	17
18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27
28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37
38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47
48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57
58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67
68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77

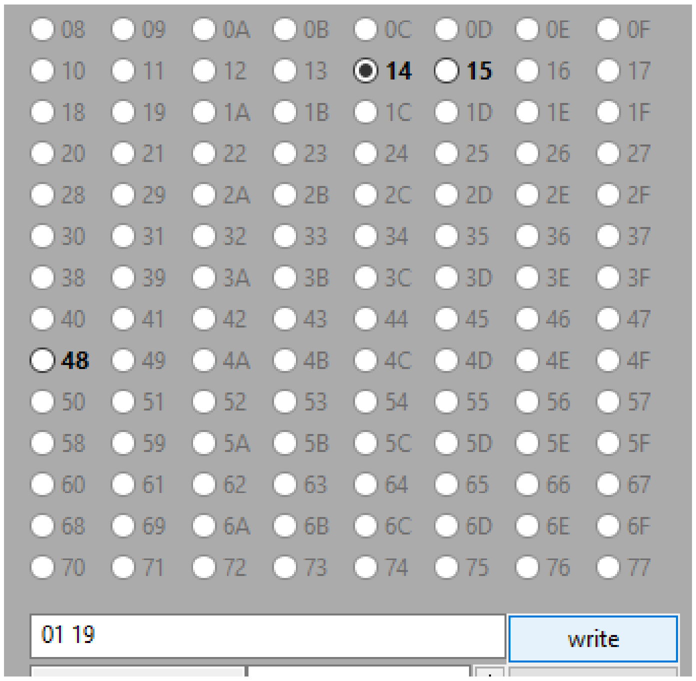
1



Selecting address `48` and clicking on **read** reads a single byte from the temperature sensor. The I²CDriver display shows the traffic immediately.



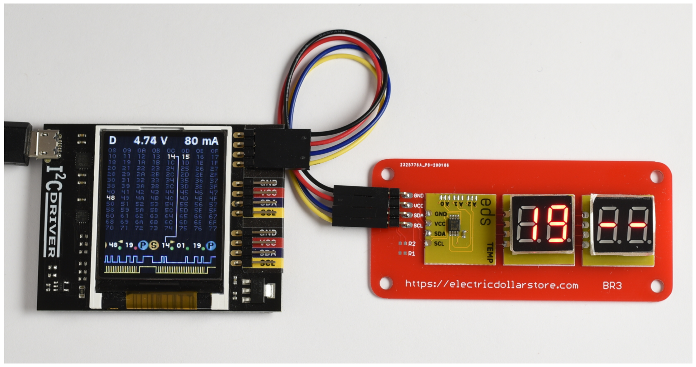
This LM75B sensor reports temperature in Celcius, so the hex byte `1A` represents a temperature of 26 °C.

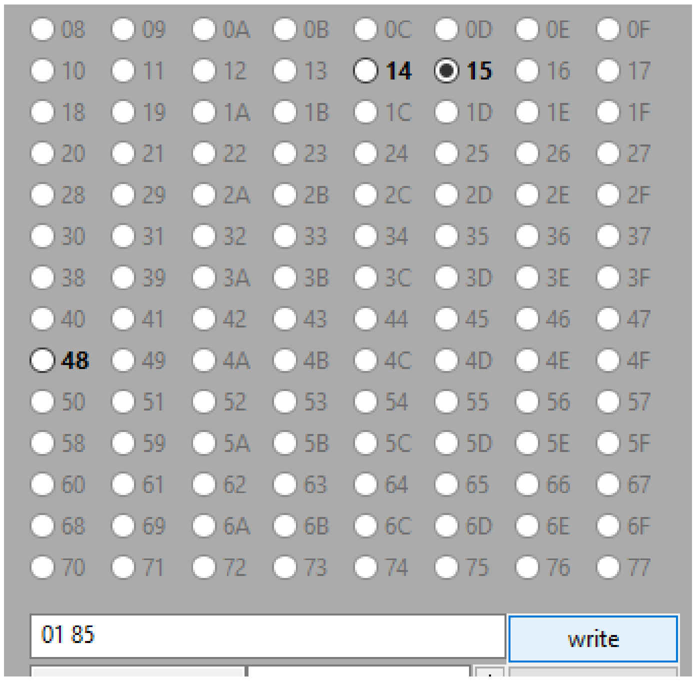


08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	[✓ **14**]	**15**	16	17
18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27
28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37
38	39	3A	3B	3C	3D	3E	3F
40	41	42	43	44	45	46	47
48	49	4A	4B	4C	4D	4E	4F
50	51	52	53	54	55	56	57
58	59	5A	5B	5C	5D	5E	5F
60	61	62	63	64	65	66	67
68	69	6A	6B	6C	6D	6E	6F
70	71	72	73	74	75	76	77

01 19

Selecting the left-hand display (address `14`) and entering the hex values `01 19` then clicking on **write** sets the first LED to the digits `19`.

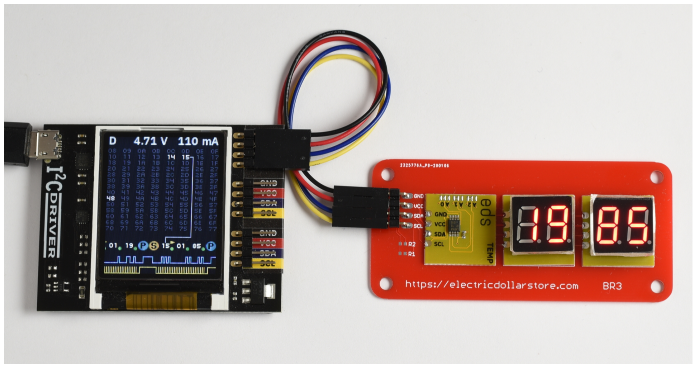




○ 08	○ 09	○ 0A	○ 0B	○ 0C	○ 0D	○ 0E	○ 0F
○ 10	○ 11	○ 12	○ 13	○ **14**	● **15**	○ 16	○ 17
○ 18	○ 19	○ 1A	○ 1B	○ 1C	○ 1D	○ 1E	○ 1F
○ 20	○ 21	○ 22	○ 23	○ 24	○ 25	○ 26	○ 27
○ 28	○ 29	○ 2A	○ 2B	○ 2C	○ 2D	○ 2E	○ 2F
○ 30	○ 31	○ 32	○ 33	○ 34	○ 35	○ 36	○ 37
○ 38	○ 39	○ 3A	○ 3B	○ 3C	○ 3D	○ 3E	○ 3F
○ 40	○ 41	○ 42	○ 43	○ 44	○ 45	○ 46	○ 47
○ **48**	○ 49	○ 4A	○ 4B	○ 4C	○ 4D	○ 4E	○ 4F
○ 50	○ 51	○ 52	○ 53	○ 54	○ 55	○ 56	○ 57
○ 58	○ 59	○ 5A	○ 5B	○ 5C	○ 5D	○ 5E	○ 5F
○ 60	○ 61	○ 62	○ 63	○ 64	○ 65	○ 66	○ 67
○ 68	○ 69	○ 6A	○ 6B	○ 6C	○ 6D	○ 6E	○ 6F
○ 70	○ 71	○ 72	○ 73	○ 74	○ 75	○ 76	○ 77

01 85

Similarly selecting address `15` and entering the hex values `01 85` then clicking on **write** sets the second LED to `85`.



5.3 The command-line tool `i2ccl`

`i2ccl` is the same on all platforms.

The first parameter to the command is the serial port, which depends on your operating system. All following parameters are control commands. These are:

`i`	display status information (uptime, voltage, current, temperature)
`d`	device scan
`w` *dev bytes*	write *bytes* to I^{2}C device *dev*
`p`	send a STOP
`r` *dev N*	read N bytes from I^{2}C device *dev*, then STOP
`m`	enter I^{2}C bus monitor mode

For example the command:

```
i2ccl /dev/ttyUSB0 r 0x48 2
```

reads two bytes from the I^{2}C device at address 0x48. So with an LM75B temperature sensor connected you might see output like:

```
0x16,0x20
```

which indicates a temperature of about 22 °C.

I^{2}C devices usually have multiple registers. To read register 3 of the LM75B, you first write the register address 3, then read two bytes as before:

```
i2ccl /dev/ttyUSB0 w 0x48 3 r 0x48 2
0x50,0x00
```

Which shows that register 3 has the value `0x5000`.

5.4 Monitor mode

In monitor mode, the I^{2}CDriver does not write any data to the I^{2}C bus. Instead it monitors bus traffic and draws it on the display. This makes it an ideal tool for troubleshooting and debugging I^{2}C hardware and software.

To show that it is in monitor mode, the I^{2}CDriver changes the character in the top-left of the display from `D` to `M`.

There are several ways of entering monitor mode:

- use the command-line tool:

```
i2ccl m
```

- from the GUI check the ”Monitor” box
- from Python issue:

```
i2c.monitor(True)
```

  and to exit:

```
i2c.monitor(False)
```

- connect a terminal to the I^{2}CDriver (at 1000000 8N1) and type the `m` character, then type any character to exit monitor mode

5.5 Capture mode

In capture mode, the I^{2}CDriver does not write any data to the I^{2}C bus. Instead it monitors bus traffic and transmits it via USB for recording on the PC.

5.5.1 Command line

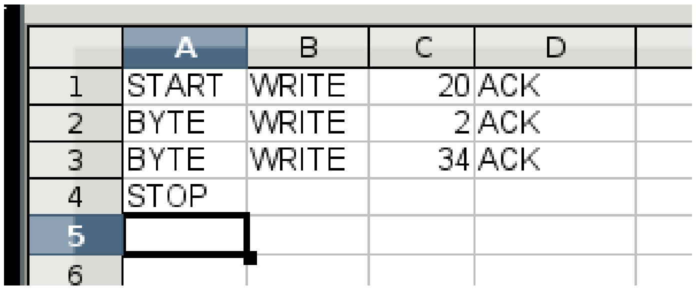
There is a Python sample program that can be used to capture traffic on the command-line at `capture.py`.

Running it with the I^{2}CDriver address as an argument puts the I^{2}CDriver into capture mode: the character in the top-left of the display changes from `D` to `C`.

```
$ python samples/capture.py /dev/ttyUSB0
```

```
Now capturing traffic to
   standard output (human-readable)
   log.csv
Hit CTRL-C to leave capture mode
<START 0x14 WRITE ACK>
<WRITE 0x02 ACK>
<WRITE 0x22 ACK>
<STOP>
^C
Capture finished
```

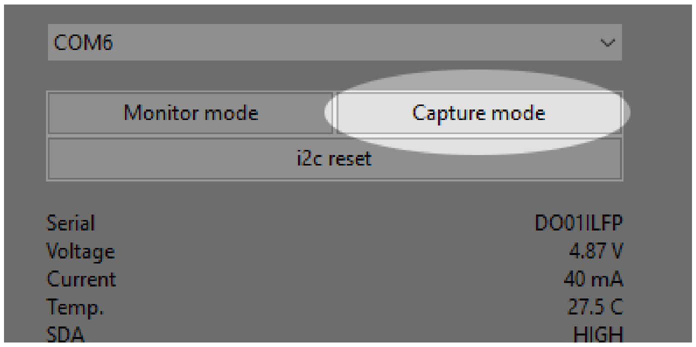
When run, it displays any traffic on standard output. It also writes a traffic summary to `log.csv` which can be examined and processed by any tool that can accept CSV files.



	A	B	C	D
1	START	WRITE	20	ACK
2	BYTE	WRITE	2	ACK
3	BYTE	WRITE	34	ACK
4	STOP			
5				
6				

5.5.2 GUI

The GUI also supports capture to CSV file.



Serial	DO01ILFP
Voltage	4.87 V
Current	40 mA
Temp.	27.5 C
SDA	HIGH

Clicking “Capture mode” starts the capture and prompts for a destination CSV file. The character in the top-left of the display changes from `D` to `C`. Capture continues until you click “Capture mode” again.

6 Examples

The Python `samples` directory contains short examples of using all Electric Dollar Store I^{2}C modules:

Module	Function	Sample
DIG2	2-digit 7-seg display	EDS-DIG2.py
LED	RGB LED	EDS-LED.py
POT	potentiometer	EDS-POT.py
BEEP	Piezo beeper	EDS-BEEP.py
REMOTE	IR remote receiver	EDS-REMOTE.py
EPROM	CAT24C512 64 Kbyte EPROM	EDS-EPROM.py
MAGNET	LIS3MDL magnetometer	EDS-MAGNET.py
TEMP	LM75B temperature sensor	EDS-TEMP.py
ACCEL	RT3000C Accelerometer	EDS-ACCEL.py
CLOCK	HT1382 real-time clock	EDS-CLOCK.py

All demos and applications are run the same way, supplying the I^{2}CDriver on the command-line. For example:

```
python EDS-LED.py COM16
```

Also included are some small applications which demonstrate combinations of modules.

6.1 Color Compass

Source code: EDS-color-compass.py

Color compass uses MAGNET and LED, reading the current magnetic field direction and rendering it as a color on the LED. As you twist the module, the color changes. For example there is a particular direction for pure red, as well as all other colors. The code reads the magnetic field direction, scales the values to 0-255, and sets the LED color.

6.2 Egg Timer

Source code: EDS-egg-timer.py

The demo uses POT, DIG2 and BEEPER to make a simple kitchen egg timer. Twisting the POT sets a countdown time in seconds, and after it's released the ticker starts counting. When it reaches “00” it flashes and beeps.

6.3 Take-a-ticket

Source code: EDS-take-a-ticket.py

This demo runs a take-a-ticket display for a store or deli counter, using REMOTE, DIG2 and BEEP modules. It shows 2-digit “now serving” number, and each time '+' is pressed on the remote it increments the counter and makes a beep, so the next customer can be served. Pressing '-' turns the number back one.

7 Technical notes

7.1 Port names

The serial port that I²CDriver appears at depends on your operating system.

On **Windows**, it appears as `COM1`, `COM2`, `COM3` etc. You can use the Device Manager or the `MODE` command to display the available ports. This article describes how to set a device to a fixed port.

On **Linux**, it appears as `/dev/ttyUSB0`, `1`, `2` etc. The actual number depends on the order that devices were added. However it also appears as something like:

```
    /dev/serial/by-id/usb-FTDI_FT230X_Basic_UART_D000QS8D-if00-port0
```

Where `D000QS8D` is the serial code of the I²CDriver (which is printed on the bottom of each I²CDriver). This is longer, of course, but always the same for a given device. You can create a symlink to refer to the device easily from scripts:

```
ln -s /dev/serial/by-id/usb-FTDI_FT230X_Basic_UART_D000QS8D-if00-port0
    ~/ex1
```

Similarly on **Mac OS**, the I²CDriver appears as `/dev/cu.usbserial-D000QS8D`.

7.2 Decreasing the USB latency timer

I²CDriver performance can be increased by setting the USB latency timer to its minimum value of 1 ms. This can increase the speed of two-way traffic by up to 10X.

On **Linux** do:

```
    setserial /dev/ttyUSB0 low_latency
```

On **Windows** and **Mac OS** follow these instructions.

7.3 Temperature sensor

The temperature sensor is located in the on-board EFM8 microcontroller. It is calibrated at manufacture to within 2 °C.

7.4 Raw protocol

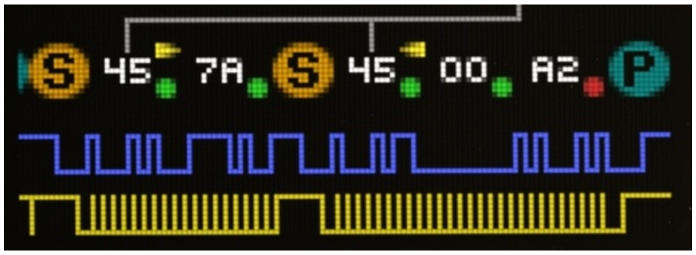
I2CDriver uses a serial protocol to send and receive I2C commands. Connect to the I2CDriver at 1M baud, 8 bits, no parity, 1 stop bit (1000000 8N1).

Because many I2CDriver commands are ASCII, you can control it interactively from any terminal application that can connect at 1M baud. For example typing `?` displays the status info, and `d` returns a bus scan.

Commands are:

`?`	transmit status info
`e`	echo byte
`1`	set speed to 100 KHz
`4`	set speed to 400 KHz
`s`	send START/addr, return status
0x80-bf	read 1-64 bytes, NACK the final byte
0xc0-ff	write 1-64 bytes
`a`	read N bytes, ACK every byte
`p`	send STOP
`x`	reset I2C bus
`r`	register read
`d`	scan devices, return 112 status bytes
`m`	enter monitor mode
`c`	enter capture mode
`b`	enter bitbang mode
`i`	leave bitmang, return to I2C mode
`u`	set pullup control lines
`_`	reboot I2CDriver

So for example to send this sequence:



Then the serial conversation is:

sender	bytes	meaning
host	`s 0x90`	START write to device 45
I²CDriver	0x01	acknowledge
host	`0xc0 0x7a`	Write 1 byte
I²CDriver	0x01	acknowledge
host	`s 0x91`	START read from device 45
I²CDriver	0x01	acknowledge
host	`0x81`	Read 2 bytes, NAK on last
I²CDriver	0x00 0xa2	byte data
host	`p`	STOP

7.4.1 `?`: transmit status info

7	6	5	4	3	2	1	0
`?` (0x3f)							

The response is always 80 bytes, space padded. For example::

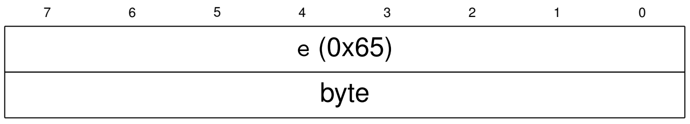
```
[i2cdriver1 DO01JUOO 000000061 4.971 000 23.8 I 1 1 100 24 ffff          ]
```

The fields are space-delimited:

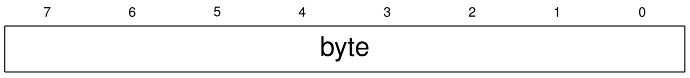
identifier	`i2cdriver1` for I^{2}CDriver, `i2cdriverm` for I^{2}CMini
serial	serial code identifier
uptime	I^{2}CDriver uptime 0-999999999, in seconds
voltage	USB bus voltage, in volts
current	attached device current, in mA
temperature	junction temperature, in °C
mode	current mode, `I` for I^{2}C, `B` for bitbang
SDA	SDA line state, 0 or 1
SCL	SCL line state, 0 or 1
speed	I^{2}C bus speed, in KHz
pullups	pullup state byte
crc	16-bit CRC of all input and output bytes (CRC-16-CCITT)

The Python sample `confirm.py` shows the CRC-16-CCITT calculation.

7.4.2 `e`: echo byte

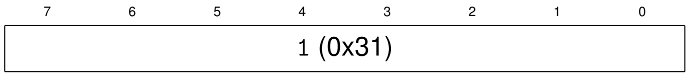


byte is any byte value. The reponse is the same value:



This command is normally used when first connecting to the I^{2}CDriver to check the serial channel.

7.4.3 `1`: set speed to 100 KHz



Sets the I²C bus speed to 100 KHz. There is no reponse.

7.4.4 `4`: set speed to 400 KHz

7	6	5	4	3	2	1	0
`4` (0x34)							

Sets the I²C bus speed to 400 KHz. There is no reponse.

7.4.5 `s`: START

7	6	5	4	3	2	1	0
`s` (0x73)							
dev							r/w

dev is the I²C 7-bit device address

r/w is 0 to start a write, 1 to start a read

The single byte response is:

7	6	5	4	3	2	1	0
0	0	1	1	0	ARB	TO	ACK

ARB is set if bus arbitration is lost during the transmission

TO is set if the transmission times out

ACK is set if the I²C device acknowledged the transmission

7.4.6 `0x80-bf` : read 1-64 bytes, NACK the final byte

7	6	5	4	3	2	1	0
1	0	count-1					

count is the number of bytes to read.

I²CDriver ACKs each read byte, except the last which is NACKed.

The response is **count** bytes of I²C bus data.

7.4.7 `0xc0-ff` : write 1-64 bytes

7	6	5	4	3	2	1	0
1	1	count-1					
byte$_0$							
...							
byte$_{count-1}$							

The single byte response is:

7	6	5	4	3	2	1	0
0	0	1	1	0	ARB	TO	ACK

ARB is set if bus arbitration is lost during the transmission

TO is set if the transmission times out

ACK is set if the I^{2}C device acknowledged the transmission

7.4.8 `a`: read N bytes, ACK every byte

7	6	5	4	3	2	1	0
`a` (0x61)							
count							

count is the number of bytes to read.

I^{2}CDriver ACKs each read byte. This command is useful when reading more than 64 bytes from an I^{2}C device.

The response is **count** bytes of I^{2}C bus data.

7.4.9 `p`: send STOP

7	6	5	4	3	2	1	0
`p` (0x70)							

Send an I^{2}C STOP symbol, ending the current transaction. There is no response.

7.4.10 `x`: reset I2C bus

7	6	5	4	3	2	1	0
`x` (0x78)							

Attempts an I2C bus reset. This consists of:

- 10 pulses of SCK with SDA high
- an I2C STOP symbol

The reponse is sent after the reset, and indicates the state of the I2C signals:

7	6	5	4	3	2	1	0
						SDA	SCL

If both signals are 1, the I2C bus is free.

7.4.11 `r`: register read

7	6	5	4	3	2	1	0
`r` (0x72)							
0	dev						
addr							
count							

dev is the I2C 7-bit device address
addr is the device register address
count is the number of bytes to read

Reads an I2C device register data. This command executes the following I2C operations:

- START, select address **dev** for writing
- write a single byte **addr**
- START, select address **dev** for reading
- read **count** bytes item STOP

The response is **count** bytes of I2C bus data.

7.4.12 `d`: scan devices, return 112 status bytes

7	6	5	4	3	2	1	0
`d` (0x64)							

Execute an I²C bus scan for devices from addresses 0x08 to 0x77.

The response is 112 bytes. Each byte is:

7	6	5	4	3	2	1	0
0	0	1	1	0	ARB	TO	ACK

ARB is set if bus arbitration is lost during the transmission

TO is set if the transmission times out

ACK is set if the I²C device acknowledged the transmission

7.4.13 `m`: enter monitor mode

7	6	5	4	3	2	1	0
`m` (0x6d)							

Enters monitor mode. The I²CDriver updates the graphical display with any traffic on the I²C bus. To exit monitor mode, send byte 0x20:

7	6	5	4	3	2	1	0
0x20							

7.4.14 `c`: enter capture mode

TBD

7.4.15 `b`: enter bitbang mode

7	6	5	4	3	2	1	0
`b` (0x62)							

Enters bitbang mode. See section 7.6 Bitbang mode.

7.4.16 `i`: leave bitmang, return to I²C mode.

7	6	5	4	3	2	1	0
`i` (0x69)							

Leaves bitbang mode and enters I²C mode. See section 7.6 Bitbang mode.

7.4.17 `u`: set pullup control lines

7	6	5	4	3	2	1	0
`u` (0x75)							
		SCL 4.7K	SCL 4.3K	SCL 2.2K	SDA 4.7K	SDA 4.3K	SDA 2.2K

Sets the I²C line pullup resistors. See section 7.5 Pull-up resistors.

7.4.18 `_`: reboot

7	6	5	4	3	2	1	0
`_` (0x5f)							

Reboots the I²CDriver. There is no response. The host should wait at least 500ms before sending the following command.

7.5 Pull-up resistors

I²CDriver has 6 programmable pull-up resistors, 3 each for SDA and SCL. 6 control bits each enable or disable a pull-up resistor. These bits are:

bit	resistor
0	2.2K to SDA
1	4.3K to SDA
2	4.7K to SDA
3	2.2K to SCL
4	4.3K to SCL
5	4.7K to SCL

At boot the two 4.7K resistors are enabled. By setting combinations of parallel resistors, a range of pull-up strengths can be achieved:

4.7K	4.3K	2.2K	pull-up strength
0	0	0	0 (i.e. no pull-up)
0	0	1	2.2K
0	1	0	4.3K
0	1	1	1.5K
1	0	0	4.7K
1	0	1	1.5K
1	1	0	2.2K
1	1	1	1.1K

Ordering this by useful resistances, the 3-bit combinations are:

3-bit value	Resistance
0	0
1	2.2K
2	4.3K
4	4.7K
5	1.5K
7	1.1K

In Python, the pullups are controlled by the `setpullups()` method, and the state can be read from the `pullups` variable. Both are 6-bit values as above.

The GUI has a control for the pull-up resistors. It sets the same pull-up strength for both SDA and SCL.

7.6 Bitbang mode

In bitbang mode, the SCL and SDA signals are driven as bidirectional IO signals.

Following the 'b' command, the host sends a **bitbang control sequence** where each sent byte controls the state of the SCL and SDA signals. Each byte is a bitfield, encoded as:

bit	meaning
0	SDA pin direction (0: input, 1: output)
1	SDA pin output value
2	SCL pin direction (0: input, 1: output)
3	SCL pin output value
4	report pin states

Note that when using a pin as an input, the corresponding output value bit should be '1'. When bit 4 is set, the pins are sampled and I^{2}CDriver transmits a byte to the host with their state:

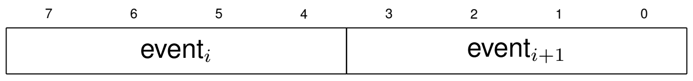
bit	meaning
0	SDA state
1	SCL state

The special value `0x40` ends the bitbang control sequence, but does not exit bitbang mode. To exit bitbang mode and re-enter I^{2}C mode, send command 'i'.

7.7 Capture mode

In capture mode I^{2}CDriver monitors the I^{2}C bus and reports all events back to the USB host, encoded in a byte stream.

Each byte contains two 4-bit event symbols packed so that the first event symbol is in the 4 more significant bits, and the second event is in the 4 less significant bits.



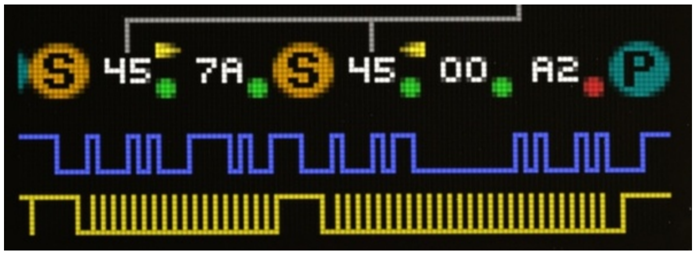
The following 4-bit event symbols are defined:

code	meaning
`0x0`	bus is idle
`0x1`	START
`0x2`	STOP
`0x8`	bit triple $0, 0, 0$
`0x9`	bit triple $0, 0, 1$
`0xa`	bit triple $0, 1, 0$
`0xb`	bit triple $0, 1, 1$
`0xc`	bit triple $1, 0, 0$
`0xd`	bit triple $1, 0, 1$
`0xe`	bit triple $1, 1, 0$
`0xf`	bit triple $1, 1, 1$

I^{2}CDriver sends the ”bus is idle” token after the bus has been idle for approximately 30 ms. Hence when there is no bus activity ”bus is idle” tokens arrive at about 30 Hz.

I^{2}C is a byte-oriented protocol, and each byte is followed by an ACK/NAK bit, so each byte actually appears as 9 bits on the wire, with the final bit as the ACK/NAK. Hence each I^{2}C byte results in three triples being sent. The first 8 bits are the big-endian I^{2}C byte. The final bit is the ACK/NAK. Note that I^{2}C represents ACK as 0, and NAK as 1.

As an example, the following wire sequence



would appear on the capture interface as the following event symbols:

code	meaning	interpretation
`0x1`	START	
`0xc`	bit triple $1, 0, 0$	
`0xa`	bit triple $0, 1, 0$	
`0xc`	bit triple $1, 0, 0$	I2C address 0x45, write, ACK
`0xb`	bit triple $0, 1, 1$	
`0xe`	bit triple $1, 1, 0$	
`0xc`	bit triple $1, 0, 0$	write byte 0x7a, ACK
`0x1`	START	
`0xc`	bit triple $1, 0, 0$	
`0xa`	bit triple $0, 1, 0$	
`0xe`	bit triple $1, 1, 0$	I2C address 0x45, read, ACK
`0x8`	bit triple $0, 0, 0$	
`0x8`	bit triple $0, 0, 0$	
`0x8`	bit triple $0, 0, 0$	read byte 0x00, ACK
`0xd`	bit triple $1, 0, 1$	
`0x8`	bit triple $0, 0, 0$	
`0xd`	bit triple $1, 0, 1$	read byte 0xa2, NAK
`0x2`	STOP	

Hence the bytes sent from the I2CDriver for this sequence are:

```
0x1c, 0xac, 0xbe, 0xc1, 0xca, 0xe8, 0x88, 0xd8, 0xd2
```

7.8 Specifications

I²CDriver DC characteristics

	min	typ	max	units
Voltage accuracy		0.01		V
Current accuracy		5		mA
Temperature accuracy		± 2		°C
SDA,SCL				
low voltage			0.6	V
high voltage	2.7		5.8	V
Output current			470	mA
Current consumption		25		mA

I²CMini DC characteristics

	min	typ	max	units
Temperature accuracy		± 2		°C
SDA,SCL				
low voltage			0.6	V
high voltage	2.7		5.8	V
Output current			270	mA
Current consumption		5		mA

AC characteristics

	min	typ	max	units
I²C speed	100		400	Kbps
Uptime accuracy		150		ppm
Uptime rollover		31.7		years
Startup time			200	ms

8 Troubleshooting

Screen is dark after connecting USB	Check USB cable and port Return for replacement
Screen is white after connecting USB	Return for replacement
Screen appears discolored	Remove protective film from screen
Port does not appear on host	Confirm that FTDI VCP drivers are installed Check USB cable and port Return for replacement
I²CDriver reports high current usage	Check target circuit for power shorts
I²CDriver reports high temperature	Check target circuit for logic shorts
I²CDriver reports USB voltage below 4.0 V	Check USB cable and port

9 Support information

Technical and product support is available at support@spidriver.com

I²CDriver is built and maintained by Excamera Labs.

Index