Contents

1 Overview 3
   1.1 Features 3

2 Getting Started 4

3 Software installation 6
   3.1 Windows 6
   3.2 Linux 8
   3.3 MacOS 9

4 APIs 10
   4.1 Python 2 and 3 10
   4.2 C/C++ 12

5 Using I2CDriver 13
   5.1 The display 13
   5.2 The GUI 15
   5.3 The command-line tool i2ccl 19
   5.4 Monitor mode 20
   5.5 Capture mode 20
      5.5.1 Command line 21
      5.5.2 GUI 21

6 Examples 23

©2021 Excamera Labs
6.1 Color Compass ........................................... 23
6.2 Egg Timer ............................................... 24
6.3 Take-a-ticket ........................................... 24

7 Technical notes ........................................... 25
  7.1 Port names ............................................. 25
  7.2 Decreasing the USB latency timer ....................... 25
  7.3 Temperature sensor .................................... 26
  7.4 Raw protocol .......................................... 27
    7.4.1 ?: transmit status info .......................... 28
    7.4.2 e: echo byte ...................................... 29
    7.4.3 i: set speed to 100 KHz .......................... 29
    7.4.4 4: set speed to 400 KHz .......................... 30
    7.4.5 s: START ........................................... 30
    7.4.6 0x80-bf: read 1-64 bytes, NACK the final byte .... 30
    7.4.7 0xc0-ff: write 1-64 bytes ........................ 31
    7.4.8 a: read N bytes, ACK every byte .................. 31
    7.4.9 p: send STOP ....................................... 31
    7.4.10 x: reset I²C bus .................................. 32
    7.4.11 r: register read ................................... 32
    7.4.12 d: scan devices, return 112 status bytes ....... 33
    7.4.13 m: enter monitor mode ........................... 33
    7.4.14 c: enter capture mode ............................ 33
    7.4.15 b: enter bitbang mode ............................ 33
    7.4.16 i: leave bitmang, return to I²C mode. .......... 34
    7.4.17 u: set pullup control lines ...................... 34
    7.4.18 : reboot .......................................... 34
  7.5 Pull-up resistors ...................................... 34
  7.6 Bitbang mode ......................................... 35
  7.7 Capture mode ........................................ 37
  7.8 Specifications ....................................... 39

8 Troubleshooting ......................................... 40

9 Support information ..................................... 40

Index ......................................................... 41
1 Overview

I²CDriver is an easy-to-use, open source tool for controlling I²C devices. It works with Windows, Mac, and Linux, and has a built-in color screen that shows a live dashboard of all the I²C 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.

I²CMini is a reduced-size version of I²CDriver suitable for use in embedded devices. It is 100% compatible with I²CDriver but lacks a display, current or voltage monitoring. In every other respect it behaves identically to I²CDriver.

1.1 Features

- **Live display**: shows you exactly what it's doing all the time
- **Supports all I²C features**: 7- and 10-bit I²C addressing, clock stretching, bus arbitration, and sustained I²C transfers at 400 and 100 kHz
- **I²C pullups**: programmable I²C 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 I²C ports**: three identical I²C ports, each with power and I²C 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:

<table>
<thead>
<tr>
<th>GND</th>
<th>black</th>
<th>signal ground</th>
</tr>
</thead>
<tbody>
<tr>
<td>VCC</td>
<td>red</td>
<td>3.3 V supply</td>
</tr>
<tr>
<td>SDA</td>
<td>blue</td>
<td>I²C data</td>
</tr>
<tr>
<td>SCL</td>
<td>yellow</td>
<td>I²C clock</td>
</tr>
</tbody>
</table>

Across the top of the display I²CDriver continuously shows the USB bus voltage and the current output.
I²CMini 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 I²CDriver.
3 Software installation

The source for all the I2CDriver 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:

![GUI shortcut](image)

Launching it brings up the control window:
If there is only one serial device, the I²CDriver 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 I²C 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
```
The COM port can be found using the `MODE` command to list connected ports. You can use the Device Manager or the `MODE` command to display the available ports.

The Windows convention for COM10 and above is that they must be prefixed by `\\\`, for example:

```
c:\>i2ccl \\.\COM11
```

See (section 7.1 "Port names") for details on how to fix a device to a port number permanently.

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
```

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-XXXXXXX i
```

(substituting your actual I²CDriver's ID for XXXXXX) 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-XXXXXXX, as explained here.
4 APIs

4.1 Python 2 and 3

The I2CDriver bindings can be installed with pip like this:

```bash
pip install i2cdriver
```

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

```python
>>> 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:

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

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

```bash
python i2cgui.py
```
which depending on your distribution looks something like this:

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

The module has extensive help strings:

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

displays the API documentation, and the Python module is documented in full at

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 I²C 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.
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 Celsius, so the hex byte 1A represents a temperature of 26 °C.
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.
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 I²C device `dev`
- `p`: send a STOP
- `r dev N`: read `N` bytes from I²C device `dev`, then STOP
- `m`: enter I²C bus monitor mode

For example the command:

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

reads two bytes from the I²C 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.

I²C devices usually have multiple registers. To write to a device register, use the `w` command. Using the same device at address 0x48, writing the two byte value 0x25,0x26 to register 2 is:

```
i2ccl /dev/ttyUSB0 w 0x48 2,0x25,0x26 p
```

To read register 3 of the LM75B, first write the register address 3, then read two bytes as above:

```
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 I²CDriver does not write any data to the I²C bus. Instead it monitors bus traffic and draws it on the display. This makes it an ideal tool for troubleshooting and debugging I²C hardware and software.

To show that it is in monitor mode, the I²CDriver 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 I²CDriver (at 1000000 8N1) and type the m character, then type any character to exit monitor mode

5.5 Capture mode

In capture mode, the I²CDriver does not write any data to the I²C 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 `I2CDriver` address as an argument puts the `I2CDriver` 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.

5.5.2 GUI

The GUI also supports capture to CSV file.
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 I²C modules:

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

All demos and applications are run the same way, supplying the I²CDriver 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. The Windows convention for COM10 and above is that they must be prefixed by \\., for example: \\.
COM11. 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_DO00QS8D-if00-port0
```

Where DO00QS8D 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_DO00QS8D-if00-port0
```

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

\( I^2\text{CDriver} \) uses a serial protocol to send and receive \( I^2\text{C} \) commands. Connect to the \( I^2\text{CDriver} \) at 1M baud, 8 bits, no parity, 1 stop bit (1000000 8N1).

Because many \( I^2\text{CDriver} \) 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 \( I^2\text{C} \) 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 bitbang, return to \( I^2\text{C} \) mode
- `u` set pullup control lines
- `-` reboot \( I^2\text{CDriver} \)

So for example to send this sequence:
Then the serial conversation is:

<table>
<thead>
<tr>
<th>sender</th>
<th>bytes</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>host</td>
<td>0x90</td>
<td>START write to device 45</td>
</tr>
<tr>
<td>I²CDriver</td>
<td>0x01</td>
<td>acknowledge</td>
</tr>
<tr>
<td>host</td>
<td>0xc0 0x7a</td>
<td>Write 1 byte</td>
</tr>
<tr>
<td>I²CDriver</td>
<td>0x01</td>
<td>acknowledge</td>
</tr>
<tr>
<td>host</td>
<td>0x91</td>
<td>START read from device 45</td>
</tr>
<tr>
<td>I²CDriver</td>
<td>0x01</td>
<td>acknowledge</td>
</tr>
<tr>
<td>host</td>
<td>0x81</td>
<td>Read 2 bytes, NAK on last</td>
</tr>
<tr>
<td>I²CDriver</td>
<td>0x00 0xa2</td>
<td>byte data</td>
</tr>
<tr>
<td>host</td>
<td>p</td>
<td>STOP</td>
</tr>
</tbody>
</table>

7.4.1 question: transmit status info

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

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

```
[i2cdriver1 DO01JU00 000000061 4.971 0 0 23.8 I 1 1 100 24 ffff ]
```

The fields are space-delimited:
identifier: i2cdriver1 for I²CDriver, i2cdriverm for I²CMini
serial: serial code identifier
uptime: I²CDriver 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 I²C, B for bitbang
SDA: SDA line state, 0 or 1
SCL: SCL line state, 0 or 1
speed: I²C 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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>e (0x65)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>byte</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>byte</td>
</tr>
</tbody>
</table>

This command is normally used when first connecting to the I²CDriver to check the serial channel.

7.4.3 1: set speed to 100 KHz

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1 (0x31)</td>
</tr>
</tbody>
</table>

Sets the I²C bus speed to 100 KHz. There is no response.
7.4.4 4: set speed to 400 KHz

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 (0x34)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

7.4.5 s: START

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>s (0x73)</td>
<td></td>
<td>dev</td>
<td>r/w</td>
</tr>
</tbody>
</table>

**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:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ARB</td>
<td>TO</td>
<td>ACK</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>count-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`count-1` byte

```
... 
```

`byte_{count-1}`

The single byte response is:

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ARB</td>
<td>TO</td>
<td>ACK</td>
</tr>
</tbody>
</table>
```

- **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

7.4.8 a: read N bytes, ACK every byte

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>a (0x61)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

`count` is the number of bytes to read.

I2CDriver ACKs each read byte. This command is useful when reading more than 64 bytes from an I2C device.

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

7.4.9 p: send STOP

```
<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>p (0x70)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Send an I2C STOP symbol, ending the current transaction. There is no response.
7.4.10 \textit{x: reset I}²\text{C} bus

Attempts an I²C bus reset. This consists of:

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

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

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

7.4.11 \textit{r: register read}

\texttt{x (0x72)}

\hspace{1cm} \begin{tabular}{c|c|c}
\hline
7 & 6 & 5 \\
\hline
\hline
0 & dev & \\
\hline
\hline
addr & \\
\hline
\hline
count & \\
\hline
\end{tabular}

\texttt{x (0x72)}

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

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

- START, select address \texttt{dev} for writing
- write a single byte \texttt{addr}
- START, select address \texttt{dev} for reading
- read \texttt{count} bytes item STOP

The response is \texttt{count} bytes of I²C bus data.
### 7.4.12 d: scan devices, return 112 status bytes

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>d (0x64)</td>
</tr>
</tbody>
</table>

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

The response is 112 bytes. Each byte is:

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>ARB</td>
<td>TO</td>
<td>ACK</td>
</tr>
</tbody>
</table>

- **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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>m (0x6d)</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0x20</td>
</tr>
</tbody>
</table>

### 7.4.14 c: enter capture mode

TBD

### 7.4.15 b: enter bitbang mode

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>b (0x62)</td>
</tr>
</tbody>
</table>

Enters bitbang mode. See section 7.6 Bitbang mode.
7.4.16 i: leave bitbang, return to \( \text{i}^2\text{C} \) mode.

Leaves bitbang mode and enters \( \text{i}^2\text{C} \) mode. See section 7.6 Bitbang mode.

7.4.17 u: set pullup control lines

Sets the \( \text{i}^2\text{C} \) line pullup resistors. See section 7.5 Pull-up resistors.

7.4.18 -: reboot

Reboots the \( \text{i}^2\text{C} \text{Driver} \). There is no response. The host should wait at least 500ms before sending the following command.

7.5 Pull-up resistors

\( \text{i}^2\text{C} \text{Driver} \) 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:

<table>
<thead>
<tr>
<th>bit</th>
<th>resistor</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2.2K to SDA</td>
</tr>
<tr>
<td>1</td>
<td>4.3K to SDA</td>
</tr>
<tr>
<td>2</td>
<td>4.7K to SDA</td>
</tr>
<tr>
<td>3</td>
<td>2.2K to SCL</td>
</tr>
<tr>
<td>4</td>
<td>4.3K to SCL</td>
</tr>
<tr>
<td>5</td>
<td>4.7K to SCL</td>
</tr>
</tbody>
</table>
At boot the two 4.7K resistors are enabled. By setting combinations of parallel resistors, a range of pull-up strengths can be achieved:

<table>
<thead>
<tr>
<th>4.7K</th>
<th>4.3K</th>
<th>2.2K</th>
<th>pull-up strength</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0 (i.e. no pull-up)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2.2K</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>4.3K</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1.5K</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4.7K</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1.5K</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2.2K</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1.1K</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>3-bit value</th>
<th>Resistance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2.2K</td>
</tr>
<tr>
<td>2</td>
<td>4.3K</td>
</tr>
<tr>
<td>4</td>
<td>4.7K</td>
</tr>
<tr>
<td>5</td>
<td>1.5K</td>
</tr>
<tr>
<td>7</td>
<td>1.1K</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>bit</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SDA pin direction (0: input, 1: output)</td>
</tr>
<tr>
<td>1</td>
<td>SDA pin output value</td>
</tr>
<tr>
<td>2</td>
<td>SCL pin direction (0: input, 1: output)</td>
</tr>
<tr>
<td>3</td>
<td>SCL pin output value</td>
</tr>
<tr>
<td>4</td>
<td>report pin states</td>
</tr>
</tbody>
</table>

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 I²CDriver transmits a byte to the host with their state:

<table>
<thead>
<tr>
<th>bit</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SDA state</td>
</tr>
<tr>
<td>1</td>
<td>SCL state</td>
</tr>
</tbody>
</table>

The special value 0x40 ends the bitbang control sequence, but does not exit bitbang mode. To exit bitbang mode and re-enter I²C mode, send command ‘i’.
7.7 Capture mode

In capture mode I²CDriver monitors the I²C 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.

```
  7  6  5  4  3  2  1  0
  event_i  event_{i+1}
```

The following 4-bit event symbols are defined:

<table>
<thead>
<tr>
<th>code</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>bus is idle</td>
</tr>
<tr>
<td>0x1</td>
<td>START</td>
</tr>
<tr>
<td>0x2</td>
<td>STOP</td>
</tr>
<tr>
<td>0x8</td>
<td>bit triple 0, 0, 0</td>
</tr>
<tr>
<td>0x9</td>
<td>bit triple 0, 0, 1</td>
</tr>
<tr>
<td>0xa</td>
<td>bit triple 0, 1, 0</td>
</tr>
<tr>
<td>0xb</td>
<td>bit triple 0, 1, 1</td>
</tr>
<tr>
<td>0xc</td>
<td>bit triple 1, 0, 0</td>
</tr>
<tr>
<td>0xd</td>
<td>bit triple 1, 0, 1</td>
</tr>
<tr>
<td>0xe</td>
<td>bit triple 1, 1, 0</td>
</tr>
<tr>
<td>0xf</td>
<td>bit triple 1, 1, 1</td>
</tr>
</tbody>
</table>

I²CDriver 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.

I²C 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 I²C byte results in three triples being sent. The first 8 bits are the big-endian I²C byte. The final bit is the ACK/NAK. Note that I²C represents ACK as 0, and NAK as 1.
As an example, the following wire sequence

![Wire sequence image]

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

<table>
<thead>
<tr>
<th>code</th>
<th>meaning</th>
<th>interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1c</td>
<td>START</td>
<td></td>
</tr>
<tr>
<td>0xac</td>
<td>bit triple 1, 0, 0</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 0, 1, 0</td>
<td></td>
</tr>
<tr>
<td>0xc0</td>
<td>bit triple 1, 0, 0</td>
<td>I²C address 0x45, write, ACK</td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 0, 1, 1</td>
<td></td>
</tr>
<tr>
<td>0xe0</td>
<td>bit triple 1, 1, 0</td>
<td></td>
</tr>
<tr>
<td>0xc0</td>
<td>bit triple 1, 0, 0</td>
<td>write byte 0x7a, ACK</td>
</tr>
<tr>
<td>0x1c</td>
<td>START</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 1, 0, 0</td>
<td></td>
</tr>
<tr>
<td>0xe0</td>
<td>bit triple 1, 1, 0</td>
<td>I²C address 0x45, read, ACK</td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 0, 0, 0</td>
<td>read byte 0x00, ACK</td>
</tr>
<tr>
<td>0xd0</td>
<td>bit triple 1, 0, 1</td>
<td></td>
</tr>
<tr>
<td>0x80</td>
<td>bit triple 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td>0xd0</td>
<td>bit triple 1, 0, 1</td>
<td>read byte 0xa2, NAK</td>
</tr>
<tr>
<td>0x2</td>
<td>STOP</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>typ</th>
<th>max</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Voltage accuracy</td>
<td>0.01</td>
<td></td>
<td></td>
<td>V</td>
</tr>
<tr>
<td>Current accuracy</td>
<td>5</td>
<td></td>
<td></td>
<td>mA</td>
</tr>
<tr>
<td>Temperature accuracy</td>
<td>± 2</td>
<td></td>
<td></td>
<td>°C</td>
</tr>
</tbody>
</table>

**SDA, SCL**

- **low voltage**: 0.6 V
- **high voltage**: 2.7 V to 5.8 V

**Output current**: 470 mA

**Current consumption**: 25 mA

### I2CMini DC characteristics

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>typ</th>
<th>max</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>Temperature accuracy</td>
<td>± 2</td>
<td></td>
<td></td>
<td>°C</td>
</tr>
</tbody>
</table>

**SDA, SCL**

- **low voltage**: 0.6 V
- **high voltage**: 2.7 V to 5.8 V

**Output current**: 270 mA

**Current consumption**: 5 mA

©2021 Excamera Labs
AC characteristics

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>typ</th>
<th>max</th>
<th>units</th>
</tr>
</thead>
<tbody>
<tr>
<td>I²C speed</td>
<td>100</td>
<td></td>
<td>400</td>
<td>Kbps</td>
</tr>
<tr>
<td>Uptime accuracy</td>
<td>150</td>
<td></td>
<td></td>
<td>ppm</td>
</tr>
<tr>
<td>Uptime rollover</td>
<td>31.7</td>
<td></td>
<td></td>
<td>years</td>
</tr>
<tr>
<td>Startup time</td>
<td>200</td>
<td></td>
<td></td>
<td>ms</td>
</tr>
</tbody>
</table>

8 Troubleshooting

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Screen is dark after connecting USB</td>
<td>Check USB cable and port</td>
</tr>
<tr>
<td></td>
<td>Return for replacement</td>
</tr>
<tr>
<td>Screen is white after connecting USB</td>
<td>Return for replacement</td>
</tr>
<tr>
<td>Screen appears discolored</td>
<td>Remove protective film from screen</td>
</tr>
<tr>
<td>Port does not appear on host</td>
<td>Check that the USB Cable is not “power only”</td>
</tr>
<tr>
<td></td>
<td>Confirm that FTDI VCP drivers are installed</td>
</tr>
<tr>
<td></td>
<td>Check USB cable and port</td>
</tr>
<tr>
<td></td>
<td>Return for replacement</td>
</tr>
<tr>
<td>I²CDriver reports high current usage</td>
<td>Check target circuit for power shorts</td>
</tr>
<tr>
<td>I²CDriver reports high temperature</td>
<td>Check target circuit for logic shorts</td>
</tr>
<tr>
<td>I²CDriver reports USB voltage below 4.0 V</td>
<td>Check USB cable and port</td>
</tr>
</tbody>
</table>

9 Support information

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

I²CDriver is built and maintained by Excamera Labs.
Index

bitbang, 35
bitbang control sequence, 35
bus scan, 10, 19

capture, 37
Capture mode, 20
capture.py, 21
COM port, 8
COM10 etc., 8, 25
CRC, 29
display, 13
drivers
  C/C++, 12
  Linux, 8
  Mac, 9
  Python, 10
  Windows, 6

Example
  Beeper, 24
  Compass, 23
  LM75B, 10, 19
  Magnetometer, 23
  Potentiometer, 24
  RGB, 23

GPIO, 35
GUI, 15

heat-map, 13
hookup, 4

jumpers, 4
Monitor mode, 20
protocol, 27
pull-ups, 34
Qwiic, 5

register read, 19
Remote control, 24

speed, 29
symbols, 13
temperature sensor, 26

uptime, 40
USB
  latency, 25
  ports, 25
Mouser Electronics

Authorized Distributor

Click to View Pricing, Inventory, Delivery & Lifecycle Information:

**Excamera Labs:**

- cs-i2cdriver-01
- cs-i2cdriver-02
- CS-I2CDRIVER-03
- CS-I2CMINI-01
- CS-I2CMINI-02
- CS-I2CMINI-03