

# **pif\_z - Raspberry Pi Zero FPGA HAT**

**version 1.0**

**Copyright 2016 Bugblat Ltd.**

June 09, 2016

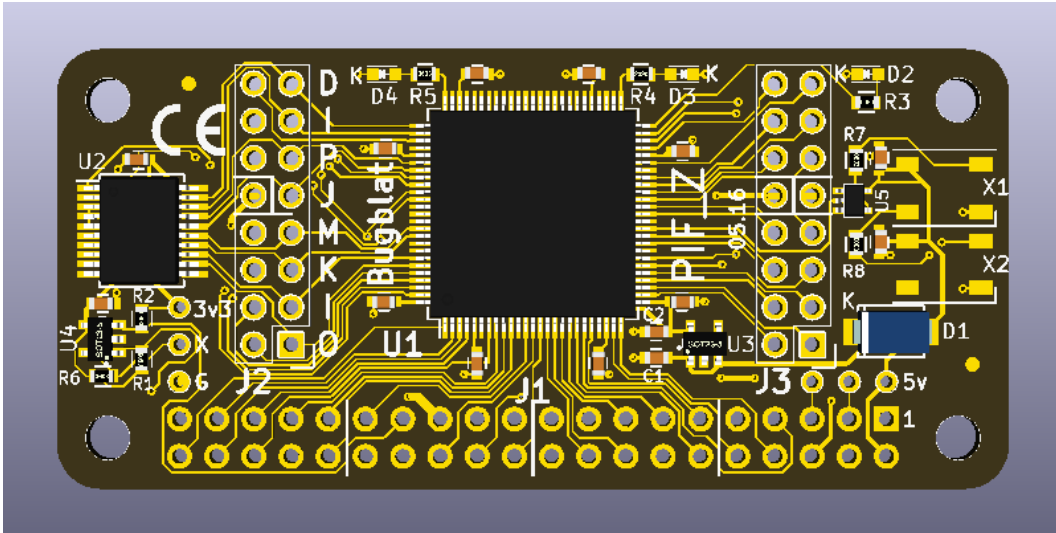


# Contents

<b>Quick Start</b>	<b>1</b>
Software Confidence Test	1
<b>Hardware</b>	<b>2</b>
Block Diagram	2
Functional Description	2
Power	3
Connectors	3
J1 - 40-pin dual-row Raspberry Pi connector	3
J2 - 16-pin eight-row expansion connector	4
J3 - 16-pin eight-row expansion connector	5
Dimensions	5
<b>Firmware</b>	<b>7</b>
Directory Structure	7
Configuration	7
flasher	8
flashctl	8
Simulating	9
Compiling	9
<b>Software</b>	<b>10</b>
Raspberry Pi Setup	10
Software Installation	11
Directory Structure	11
C/C++ shared library	11
Python Programs	12
piffind.py	12
pifload.py	12
pifweb.py	13
<b>Schematic</b>	<b>14</b>
<b>Legal Stuff</b>	<b>14</b>
The Design	14



## Quick Start



This is the documentation for Bugblat's PIF\_Z Raspberry Pi Zero FPGA HAT.

Your PIF\_Z board comes with a small application already installed - it flashes the onboard red and green LEDs in antiphase and drives random patterns to the RGB LEDs.

So all you need to do is plug your PIF\_Z board into your Pi and the LEDs should start doing what LEDs do best.

## Software Confidence Test

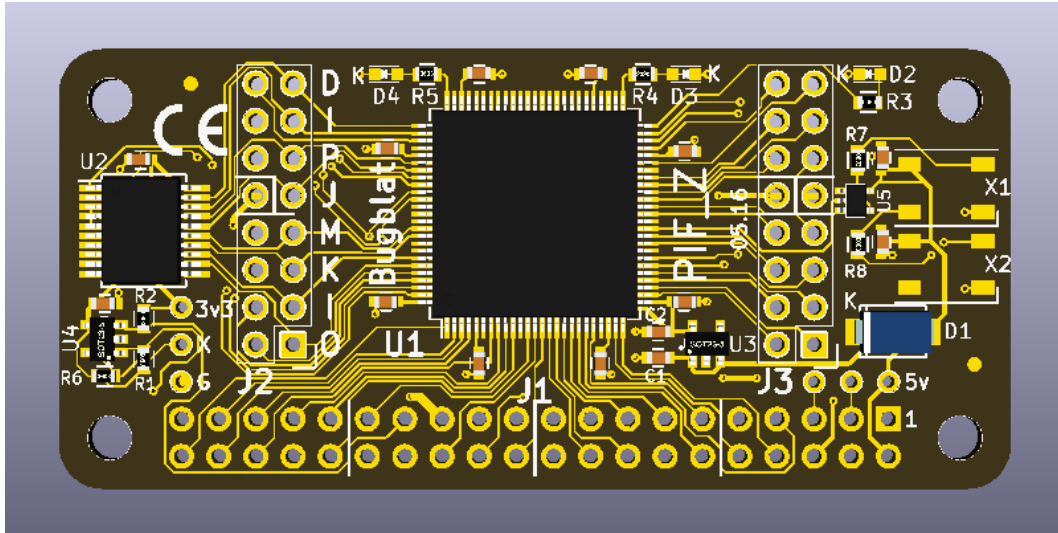
You can verify the software by flipping the PIF\_Z board's configuration firmware from the *flasher* build, where the LEDs light up in antiphase, to the *flashctl* configuration, where the LEDs light up in phase.

The Software page shows you how to enable I2C and SPI access on your Raspberry Pi Zero, and how to download the software.

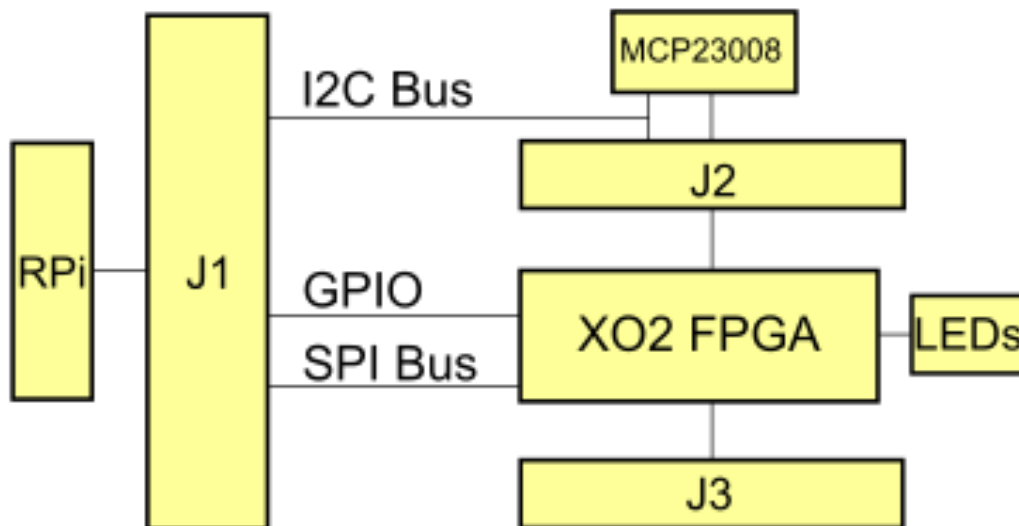
Then change to the software directory and load the *flashctl* configuration. The command line is:

```
sudo python pifload.py ../firmware/flashctl/syn/pif_flashctl.jed
```

# Hardware



## Block Diagram



## Functional Description

The key components of the Bugblat *pifZ* board are

- XO2: a Lattice Semiconductor MachXO2 FPGA (details at [Lattice](#))
- MCP23008: a Microchip I2C port expander (details at [Microchip](#)).
- 24LC32A: a Raspberry Pi *HAT* ID EEPROM (details at [Microchip](#)).
- WS2812B: two smart RGB LEDs (details at [Adafruit](#)).

The 2000HC FPGA contains 2112 four-input lookup tables (LUTs), 74Kbits of on-chip memory, and two PLLs, plus hard-wired I2C and SPI master/slave interfaces.

The MCP23008 is an 8-bit I/O expander that sits on the Raspberry Pi's I2C bus and drives the XO2's control pins. The Raspberry Pi expansion bus does not have an abundance of pins, so it makes sense to control low-activity pins in this way. The MCP23008 drives the following XO2 pins, as shown on the *pifZ* board:

- JTAGENn. This pin lets your application use the XO2 JTAG port pins when it is high, reverting the pins to JTAG usage when it is low.

## Connectors

- JTAG I/O, four pins.
- the FPGA PROGn, INITn, and DONE pins.

MCP23008 Function	Connection
GP0	FPGA pin 95 (TDO) and J2 pin 1
GP1	FPGA pin 94 (TDI) and J2 pin 3
GP2	FPGA pin 91 (TCK) and J2 pin 5
GP3	FPGA pin 90 (TMS) and J2 pin 7
GP4	FPGA pin 82 (JTAGENn) and J2 pin 9
GP5	FPGA pin 81 (PROGn) and J2 pin 11
GP6	FPGA pin 77 (INITn) and J2 pin 13
GP7	FPGA pin 76 (DONE) and J2 pin 15

## Power

The XO2 runs at 3.3V, provided by an on-board regulator connected to the Raspberry Pi's 5V pins. There is no connection to the Raspberry Pi's low-current 3.3V pins.

Regulated 3.3V is also routed to a test point adjacent to J2, but should only be used for minimal loads. For higher loads, the raw 5V from the Raspberry Pi connector is fed to a test point near J1, pin 1 (marked on the board with 5V) and you can use this to derive more current at 3.3V. This test point is a full size hole; it is also on the same 0.1" grid as J1/J2/J3.

Notice, and this is **very important**, that the pins on the XO2 can tolerate **3.3V only**. **XO2 I/O pins can not tolerate 5V**.

So the standard hookup is this:

- 5V comes in from the Raspberry Pi
- an on-board regulator drops the 5V to 3.3V
- both the 5V and the 3.3V are fed to expansion points

## Connectors

### J1 - 40-pin dual-row Raspberry Pi connector

Pin 1 is indicated by a square pad and shown on the board. As is standard, this connector has 20 rows, with two pins in each row. The pins in the first row are numbered 1 and 2, the pins in the second row are numbered 3 and 4, and so on. This is different to the traditional numbering scheme for integrated circuits.

The I2C bus runs from the Raspberry Pi to the FPGA and the MCP23008. The I2C lines (SCL and SDA) are pulled up on the Raspberry Pi board.

A second Raspberry Pi I2C bus is used to access the configuration ROM on the PIF.

The SPI bus runs from the Raspberry Pi to the FPGA. The Raspberry Pi sources two SPI *Slave Select* signals. CE0 is connected to the FPGA Sn pin and is the SPI select signal for configuration. CE1 is connected to FPGA pad 3 and is the SPI select signal for user logic.

Pins 1 to 26 match the expansion connector on the original Raspberry Pi.

Pin	Definition
1	no connection
2	5V input from the Pi
3	I2C SDA - FPGA pin 85 and MCP23008

## Connectors

4	5V input from the Pi
5	I2C SCL - FPGA pin 86 and MCP23008
6	Ground
7	Raspberry Pi clock - FPGA pin 20
8	GPIO14 - FPGA pin 25
9	Ground
10	GPIO15 - FPGA pin 24
11	GPIO17 - FPGA pin 21
12	GPIO18 - FPGA pin 19
13	GPIO27 - FPGA pin 18
14	Ground
15	GPIO22 - FPGA pin 17
16	GPIO23 - FPGA pin 16
17	no connection
18	GPIO24 - FPGA pin 15
19	SPI MOSI - FPGA pin 49
20	Ground
21	SPI MISO - FPGA pin 32
22	GPIO25 - FPGA pin 14
23	SPI SCK - FPGA pin 31
24	SPI CE0 - FPGA pin 48
25	Ground
26	SPI CE1 - FPGA pin 13
27	Configuration EEPROM I2C SDA line
28	Configuration EEPROM I2C SCL line
29	GPIO5 - FPGA pin 12
30	Ground
31	GPIO6 - FPGA pin 9
32	GPIO12 - FPGA pin 10
33	GPIO13 - FPGA pin 8
34	Ground
35	GPIO19 - FPGA pin 4
36	GPIO16 - FPGA pin 7
37	GPIO26 - FPGA pin 2
38	GPIO20 - FPGA pin 3
39	Ground
40	GPIO21 - FPGA pin 1

### ***J2 - 16-pin eight-row expansion connector***

The column of odd-numbered pins (1, 3, 5, 7, 9, 11, 13, 15) connects to the MC23008, as detailed above.



## Dimensions

Pin 1 is indicated by a square pad.

Pin	Definition
1	FPGA pin 95 (TDO)
2	FPGA pin 96
3	FPGA pin 94 (TDI)
4	FPGA pin 97
5	FPGA pin 91 (TCK)
6	FPGA pin 98
7	FPGA pin 90 (TMS)
8	FPGA pin 99
9	FPGA pin 82 (JTAGENn)
10	Ground
11	FPGA pin 81 (PROGn)
12	FPGA pin 84
13	FPGA pin 77 (INITn)
14	FPGA pin 83
15	FPGA pin 76 (DONE)
16	FPGA pin 78

### ***J3 - 16-pin eight-row expansion connector***

Pin 1 is indicated by a square pad.

Pin	Definition
1	FPGA pin 28
2	FPGA pin 27
3	FPGA pin 30
4	FPGA pin 29
5	FPGA pin 34
6	FPGA pin 35
7	FPGA pin 39
8	FPGA pin 38
9	Ground
10	Ground
11	FPGA pin 40
12	FPGA pin 41
13	FPGA pin 42
14	FPGA pin 43
15	FPGA pin 45
16	FPGA pin 47

## Dimensions

## Dimensions

- Length: 65mm (2.5 inch)
- Width: 30mm (1.2 inch)
- Thickness: standard 1.6mm PCB, plus 2mm components
- Weight: almost nothing

## Firmware

These example VHDL firmware programs are supplied with a PIF\_Z board:

1. *flasher.vhd* is a simple program that alternately flashes the red and green LEDs.
2. *flashctl.vhd* also flashes the LEDs, but in this case the flash pattern can be controlled by an external computer.

## Directory Structure

- firmware
  - pifz
    - flasher
    - flashctl
  - common

With one exception (see the Configuration section) HDL code is in the *common* directory.

## Configuration

Designs are configured for the X02-2000HC FPGA via the *pifcfg* package in *pifcfg.vhd* files in the *pifz* directory. For example:

```
-- pifcfg.vhd, PIF_Z version
--
-- Initial entry: 01-Mar-15 te
-- non-common definitions to personalise the pif implementations
--
-----
library ieee;
                                use ieee.std_logic_1164.all;

package pifcfg is

  -- PIF_Z ID = 44h = 'D'
  constant PIF_ID      : std_logic_vector(7 downto 0) := x"44"; -- 'D'
  constant X02_DENSITY : string                       := "2000L";

end package pifcfg;

-----
package body pifcfg is
end package body pifcfg;
```

Additional constants and functions can be added as a design requires. Usually the simplest practice is to define a constant in *pifcfg.vhd* and use the constant to determine properties in a lower module. For instance, a lower level module could include something like:

```
function myParameter(density: string) return integer is
begin
  if density="2000L" then
    return 1;
  else
    return 3;
  end if;
end;
```

Overall configuration definitions and useful constants are defined in the *defs* module *pifdefs.vhd* in the *common* directory. A small snip of this file is:

```

library ieee;                use ieee.std_logic_1164.all;
                             use ieee.numeric_std.all;
library work;                use work.pifcfg.all;

package defs is

  -- save lots of typing
  subtype slv2  is std_logic_vector( 1 downto 0);
  subtype slv3  is std_logic_vector( 2 downto 0);
  subtype slv4  is std_logic_vector( 3 downto 0);
  subtype slv5  is std_logic_vector( 4 downto 0);
  subtype slv6  is std_logic_vector( 5 downto 0);
  subtype slv7  is std_logic_vector( 6 downto 0);
  subtype slv8  is std_logic_vector( 7 downto 0);
  subtype slv16 is std_logic_vector(15 downto 0);
  subtype slv32 is std_logic_vector(31 downto 0);

  -----
  -- these constants are defined in outer 'pifcfg' files
  constant ID           : std_logic_vector(7 downto 0) := PIF_ID;
  constant DEVICE_DENSITY : string                    := X02_DENSITY;

  -- I2C interface -----

  constant A_ADDR      : slv2 := "00";
  constant D_ADDR      : slv2 := "01";

  constant I2C_TYPE_BITS : integer := 2;
  constant I2C_DATA_BITS : integer := 6;

```

*pifz.lpf* in the *common* directory is shared by all designs. In the main it defines the pinout of the FPGA.

## flasher

*flasher* is a straightforward design. It uses the FPGA's built in oscillator to drive PWM patterns to the on-board red and green LEDs. The LEDs are driven in antiphase.

The built in oscillator can be set to a variety of frequencies. We choose 26.6MHz, a frequency which is useful in more complex designs.

*flasher.vhd* is a wrapper, the main work is done in *piffila.vhd*.

## flashctl

*flashctl* is more complex than *flasher* - it can be controlled from the Raspberry Pi.

As before *piffila.vhd* generates antiphase LED pulses. However, the pulse stream fed to the FPGA I/Os is controlled by a register that can be written to or read from via the I2C bus.

This is how it works. The i2c stream from the Raspberry Pi is wired up to a hard coded *embedded function block* (EFB) in the FPGA.

The FPGA EFB implements:

- two i2c cores, a primary core and a secondary core
- one SPI core
- one 16-bit timer/counter
- an interface to on-chip flash memory which includes:
  - user flash memory (UFM)
  - configuration logic flash memory

## Simulating

- an interface to dynamic PLL settings
- an interface to the on-chip power controller

The EFB is exhaustively documented in the XO2 handbook which can be downloaded from the [Lattice web site](#).

Our i2c stream is connected to the EFB's *primary* i2c core. The other side of the the EFB presents a Wishbone interface to FPGA internal logic and that is the interface we use to control our logic.

The Wishbone interface is easily handled by a state machine, as seen in *pifwb.vhd*. This state machine listens to events on the Wishbone interface, and generates a minimal internal address and data bus. Here is the definition of the incoming address and data bus, extracted from *pifdefs.vhd*:

```
type XIrec is record          -- write data for regs
  PWr      : boolean;        -- registered single-clock write strobe
  PRWA     : TXA;            -- registered incoming addr bus
  PRdFinished : boolean;    -- registered in clock PRDn goes off
  PRdSubA   : TXSubA;       -- read sub-address
  PD       : TwrData;       -- registered incoming data bus
end record XIrec;
```

*pifctl.vhd* listens to this bus, writes values into registers, and reads values from registers.

Here is an example of writing to a register on a pif board:

1. the Raspberry Pi executes an i2c write to send the data over i2c to the FPGA's EFB
2. the FPGA state machine detects *data available* on the Wishbone interface, reads in the data and generates a write strobe
3. *pifctl.vhd*, or other application, logic detects the write strobe, checks for an address match, and loads the data into an internal register

So where does the internal address come from? This design splits incoming bytes into a two bit *type* field and a six bit *data* field. The *type* field can indicate an A byte or a D byte. If it is an A byte, the data field is loaded into an address register, with the six bit field allowing up to 64 addresses. If it is a D byte, the six bit data field and a write strobe go out over the internal data bus.

Reading from a register is simpler. Read data is always eight bits, there is no need for an address field in readback data. The address register is loaded just the same as for a write. A read *subaddress* is cleared to zero at the same time the address is written. The subaddress is incremented with every read.

Assuming the address has already been loaded, here is an example of reading from a register on a pif board:

1. the Raspberry Pi executes an i2c read of the FPGA's EFB
2. the FPGA state machine detects *data required* on the Wishbone interface. It writes the register data to the wishbone interface, generates a *read finished* internal strobe, and increments the subaddress
3. the Raspberry Pi picks up the data from i2c

## Simulating

Most of the design time with HDLs is spent in a simulator. *flashctl\_tb* in the *common* directory is a simulation testbed.

## Compiling

The *Lattice Diamond* system compiles HDL files to JEDEC bit streams. There are many paths for injecting the JEDEC data into a pif FPGA, but the documentation can be confusing. The official route is via the *ispUFW* and *ispVM* system.

Since a pif board is a single chip system, we can use a simple solution - the Lattice Diamond JEDEC can be loaded directly into a pif FPGA via the *pifload.py* script.

## Software

The software supplied with a pif board supports

- finding the pif board in your system
- loading a configuration into the pif board
- interacting with the pif board via a web/HTML front end

Low level functions are supplied as C/C++ programs, high level functions are in [Python](#).

## Raspberry Pi Setup

Setting up your Raspberry Pi for GPIO access, I2C, and SPI is covered by many articles on the web so we will only give a brief summary.

Enable I2C and SPI via the *raspi-config* menu under *Advanced Options*. First the main configuration menu:

```
Raspberry Pi Software Configuration Tool (raspi-config)

1 Expand Filesystem           Ensures that all of the SD card s
2 Change User Password        Change password for the default u
3 Enable Boot to Desktop/Scratch Choose whether to boot into a des
4 Internationalisation Options Set up language and regional sett
5 Enable Camera               Enable this Pi to work with the R
6 Add to Rastrack             Add this Pi to the online Raspber
7 Overclock                   Configure overclocking for your P
8 Advanced Options            Configure advanced settings
9 About raspi-config          Information about this configurat

                                <Select>                                <Finish>
```

then enable both I2C and SPI access via the Advanced Options menu:

```
Raspberry Pi Software Configuration Tool (raspi-config)

A1 Overscan                   You may need to configure oversca
A2 Hostname                   Set the visible name for this Pi
A3 Memory Split               Change the amount of memory made
A4 SSH                         Enable/Disable remote command lin
A5 Device Tree                Enable/Disable the use of Device
A6 SPI                         Enable/Disable automatic loading
A7 I2C                         Enable/Disable automatic loading
A8 Serial                     Enable/Disable shell and kernel m
A9 Audio                       Force audio out through HDMI or 3
A0 Update                      Update this tool to the latest ve

                                <Select>                                <Back>
```

then reboot (`sudo reboot`).

Download the `i2ctools` utility:

```
sudo apt-get install python-smbus
sudo apt-get install i2c-tools
```

and check that the pif board is visible:

```
sudo i2cdetect -y 1
```

With the `pif_flasher` configuration loaded you should see this:

```
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          ---
10: ---
20: 20 ---
30: ---
40: ---
50: ---
60: ---
70: ---
```

And with the `pif_flashctl` configuration loaded you should see this:

```
    0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          ---
10: ---
20: 20 ---
30: ---
40: 40 41 -- 43 ---
50: ---
60: ---
70: ---
```

20h is the I2C address for the pif's MCP23008, 40h is the XO2's I2C configuration address, 41h is the XO2's I2C user-level slave address, and 43h is the XO2's I2C state machine reset address.

The standard pif software reloads the XO2 FPGA configuration flash via an SPI configuration port, but the I2C configuration port is still active until it is explicitly disabled.

## Software Installation

The software can be downloaded from <http://www.bugblat.com/products/pifz/pif.zip>.

Alternatively you can download a Git repo: <https://github.com/bugblat/pifz>

## Directory Structure

- src
- static
- templates

## C/C++ shared library

To control your pif board you need to

- access the Raspberry Pi SPI and I2C pins
- control the pif's onboard MCP23008
- control the pif's FPGA

To ease this task we provide shared library - `libpif.so`. `libpif` is written in C++, with a C wrapper so that it can interface easily to scripting languages such as Python. The source files are in the `src` directory. The interface is defined in the `pifwrap.h` file.

For SPI and I2C access on the Raspberry Pi, we use Mike McCauley's [BCM2835](#) GPIO library.

To compile and install `libpif.so` you need to change to the `software/src` directory and enter the usual recipe:

```
make
sudo make install
```

The software/src directory includes a precompiled libpif.so file - you can run the Python software even if the compiler tools for C/C++ are missing from your system. You still need to run the install step (sudo make install) if you use the precompiled libpif.so.

We use the ctypes package for the interface between libpif.so and Python scripts.

## Python Programs

All the Python programs are provided as uncompiled files. Because they access SPI and I2C GPIO pins, they must be run with root privileges, most easily via the sudo command. For example:

```
sudo python piffind.py
```

### piffind.py

This program scans the SPI bus. Here is the output from a run on my computer:

```
===== pif find =====
Using pif library version: 'libpif,Jun  9 2016,15:36:37'

X02 Device ID: 012bb043 - device is an X02-2000HC

===== bye =====
```

### pifload.py

This program takes a configuration JEDEC file as input. It then

1. searches for a pif board
2. clears the pif's FPGA flash memory
3. loads the new configuration data into the flash memory
4. reinitializes the FPGA.

For example, with this command line:

```
sudo python pifload.py pif_flasher.jed
```

this is the output from a run on my computer (the line starting *programming* has been shortened):

```
=====hello=====
Configuration file is pif_flasher.jed
Using pif library version: 'libpif,Jun  9 2016,15:36:37'

X02 Device ID: 012bb043 - device is an X02-2000HC
X02 Trace ID : 00.44.30.96_43.04.22.09
X02 usercode from Flash:  00.00.00.00
X02 usercode from SRAM :  50.49.46.30
JEDEC file is pif_flasher.jed
starting to read JEDEC file
first configuration data line: 23
. . . . .
last configuration data line: 574
552 frames
finished reading JEDEC file
erasing configuration flash ... erased
programming configuration flash ... . . . . . programmed
transferring ...
configuration finished.

===== bye =====
```



## pifweb.py

This program implements browser control of the flashctl configuration in a pif board. You need to have installed Aaron Swartz' [web.py](#) script:

```
sudo apt-get install python-webpy
```

There are several parts:

- *pifweb.py* uses *web.py* to
  - start a web server
  - serve up the application web page
  - listen for *GET* and *POST* commands from the web page
  - communicate with the pif board
  - send replies to the web page
- the content of the HTML that is generated is governed by *index.html*, *layout.html*, *header.html*, and *footer.html* in the *templates* directory
- the appearance of the HTML is governed by *style.css* in the *static* directory

Make sure you have loaded the *flashctl* configuration in your pif board, for example via this command line:

```
sudo python pifload.py pif_flashctl.jed
```

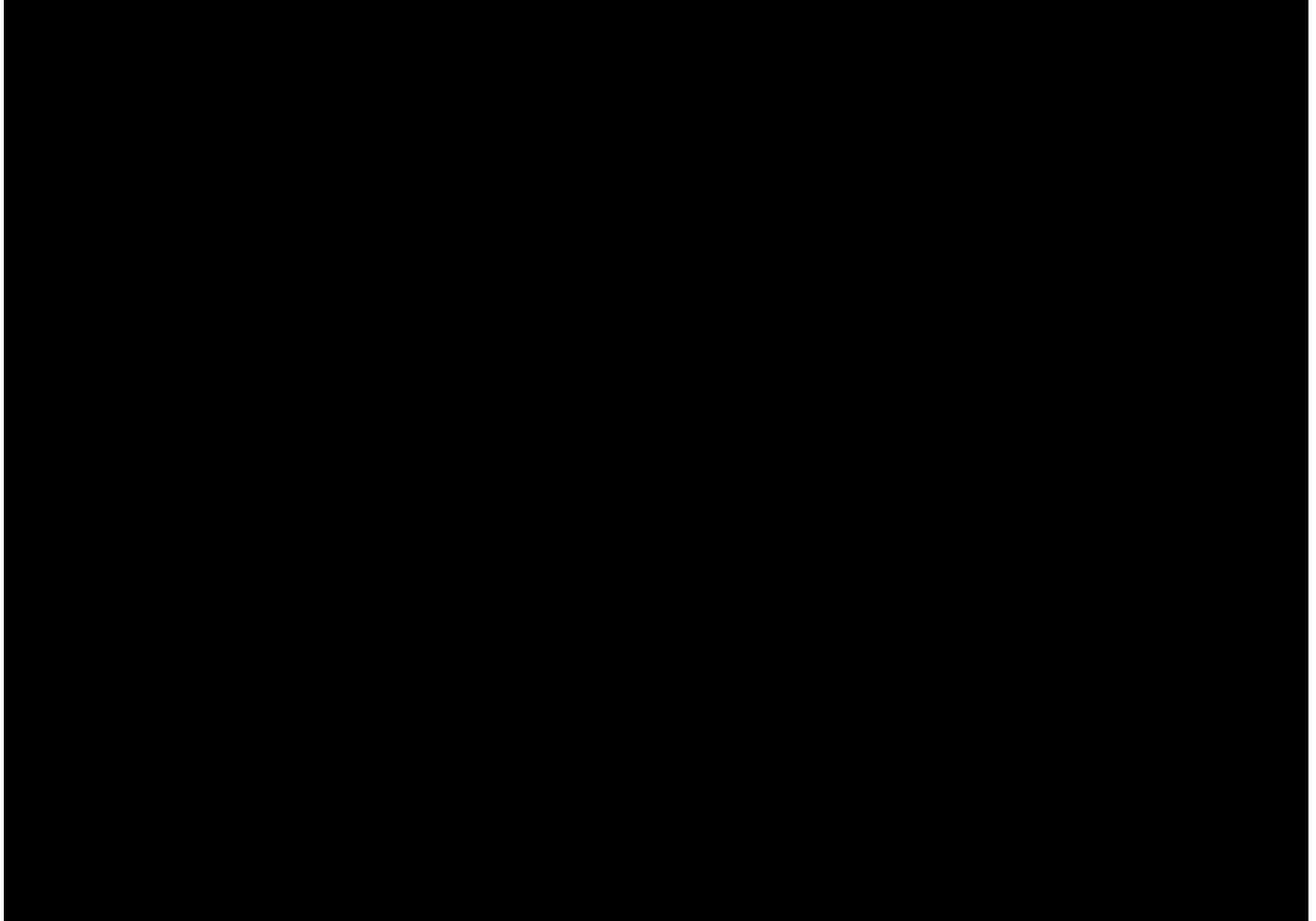
Start the program in a command window:

```
sudo python pifweb.py
```

Then point your browser at localhost:8080. This is what my browser shows:



## Schematic



## Legal Stuff

This is a board for inquisitive minds with a basic understanding of electronics. You know what that means.

Since the board is not a completed product it may not meet all the regulatory and safety compliance standards which may normally be associated with similar items. You assume full responsibility to determine and/or assure compliance with any such standards and related certifications as may be applicable. You will employ reasonable safeguards to ensure that your use of the the board will not result in any property damage or injury or death, even if the the board should fail to perform as described or expected.

## The Design

The design materials referred to in this document are **not supported** and do **not** constitute a reference design.

**To the extent permitted by applicable law there is no warranty for the design materials. Except when otherwise stated in writing the copyright holders and/or other parties provide the design materials as is without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the design materials is with you. Should the design materials prove defective, you assume the cost of all necessary servicing, repair or correction.**

This board was designed as an evaluation and development tool. It was not designed with any other application in mind. As such, these design materials may or may not be suitable for any other

## Schematic

purposes. If any design material is used it becomes your responsibility as to whether it meets your specific needs or the needs of your specific applications and the design material may require changes to meet your requirements.