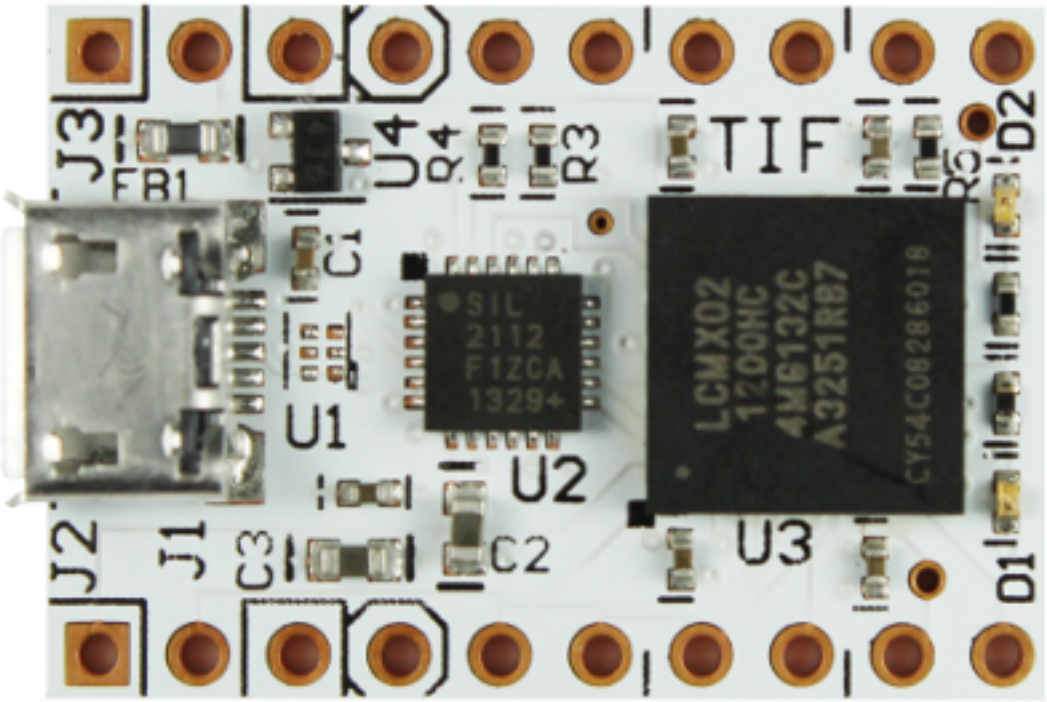
tif - A Tiny FPGA Board

version 1.0

October 03, 2013

Contents

Quick Start



Just plug it in! A tif board connects with the USB HID (Human Interaction Device) protocol, so it does not need any special drivers. It also comes with an small application already installed - it flashes the onboard LEDs in antiphase.

So plug your tif board into a USB micro lead (micro is the type of USB lead used in most modern phones, pads, and ereaders) and the LEDs should start doing what LEDs do best. There may be a small delay while the computer's operating system loads its built in HID driver.

Software Confidence Test

You can verify the software by flipping the tif 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.

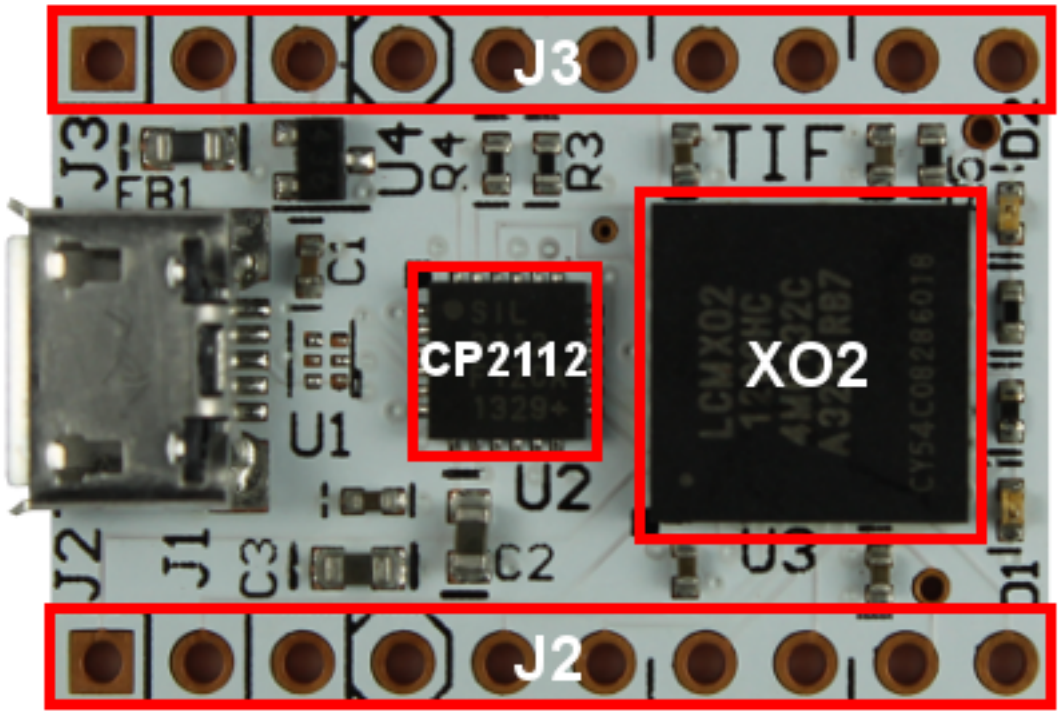
Download the software as described in the *Software* page and change to the software directory. If you are using Linux you will also have to run the setup and build scripts as described in the *Software* page. Load the *flashctl* configuration into the tif. If you have a tif-1200, the command line is:

```
python tifload.py ../firmware/1200/flashctl/syn/tif_flashctl.jed
```

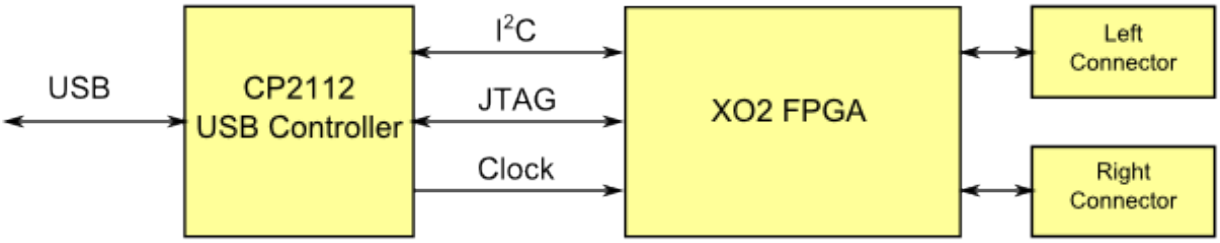
If you have a tif-4000, the command line is:

```
python tifload.py ../firmware/4000/flashctl/syn/tif_flashctl.jed
```

Hardware



Block Diagram



Functional Description

The key components of the Bugblat *tif* board are

- XO2: a Lattice Semiconductor MachXO2 FPGA (details at Lattice)
- CP2112: a Silicon Labs CP2112 USB/HID slave controller (details at SiLabs).

A tif-1200 uses an XO2-1200HC FPGA and a tif-4000 uses an XO2-4000HC FPGA. The 1200HC FPGA contains 1280 four-input lookup tables (LUTs), the 4000HC FPGA contains 4320 LUTs. The 4000HC part also has more on-cip memory and a second PLL. More LUTs means that a more complex design can be fitted into the FPGA.

The CP2112 is a hard programmed microcontroller that provides a USB HID client service. Under the covers it is an OTP 8051 variant, but that is not visible. The CP2112 listens to HID requests over a USB connection and drives the following pins:

- an i2c connection to both the XO2 and the expansion connector
- a clock, connected to the XO2. The clock is derived from locking an on-chip PLL to the USB clock,
- JTAGENn, connected to the XO2. 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.

- JTAG I/O, connected to the XO2.
- two spare pins, connected to the XO2
- a *suspend* signal that goes high when the USB connection is suspended. You could use this signal to flip your application into a low power mode when USB suspends.

Pin	Function	Connection
24	SCL	FPGA pin C8. Also J2 - see below.
1	SDA	FPGA pin B8. Also J2 - see below.
23	GPIO_0	FPGA pin A13, spare
22	GPIO_1	FPGA pin A12, spare
21	GPIO_2	FPGA pin B9, JTAGEN
20	GPIO_3	FPGA pin B6, TCK
15	GPIO_4	FPGA pin A6, TMS
14	GPIO_5	FPGA pin A4, TDO
13	GPIO_6	FPGA pin B4, TDI
12	GPIO_7	FPGA pin C1, PLL input, usually a 24MHz clock
11	SUSPND	FPGA pin E3

Power

The XO2 runs at 3.3V, provided by the CP2112's on-chip regulator. Maximum current from this regulator is 100mA, enough for the low power XO2.

Regulated 3.3V is also routed to J2, but should only be used for minimal loads. For higher loads, the raw 5V from the USB connector is fed out to J3 and you can use this to derive more current at 3.3V. But 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:

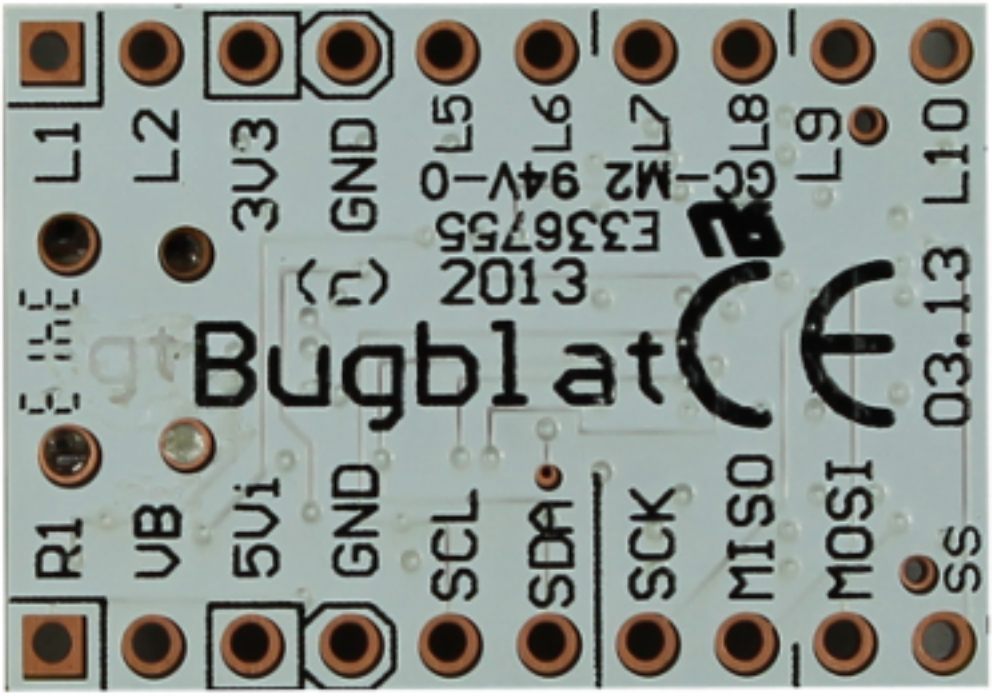
- 5V comes in from USB
- the CP2112 regulates the 5V to 3.3V
- both the 5V and the 3.3V are fed to expansion connector pins

Your tif will still work if is not hooked up to a USB connection, you just need to provide a regulated 5V input. You can do this in either of these ways:

- connect regulated 5V into the USB connector from a phone or pad charger or from a phone backup pack
- connect regulated 5V to the *5V input* on J3. see the schematic for how this works.

Connectors

This is the back view of a tif board. J2 is at the top.



Left (J2)

This is the left connector when the board is viewed from the component side with the USB connector at the top.

Pin	Definition
1	FPGA pin A2
2	FPGA pin B1
3	3.3V from the CP2112 regulator. Minimal current available
4	Ground
5	FPGA pin E1
6	FPGA pin F1
7	FPGA pin H1
8	FPGA pin J1
9	FPGA pin K1 and test point TP1
10	FPGA pin M1

Right (J3)

This is the right connector when the board is viewed from the component side with the USB connector at the top.

Pin	Definition
1	FPGA pin B14
2	USB VBUS
3	5V input
4	Ground
5	I2C SCL - FPGA pin B8 (also FPGA pins B7 and A7 and CP2112)

6	I2C SDA - FPGA pin C8 (also CP2112)
7	SPI SCK - FPGA pin M4
8	SPI MISO - FPGA pin N4 (also FPGA pin P4)
9	SPI MOSI - FPGA pin P13
10	FPGA pin P2

The I2C lines (SCL and SDA) are pulled up with 2.2Kohm to 3.3V.

There are three test points on a tif board:

- TP1 is connected to J2 pin 9.
- TP2 is connected to the FPGA's *SPI slave* configuration select pin. See the XO2 handbook for more details.
- TP3 is connected to the VPP pin on the CP2112. See the CP221 data sheet for more details.

Dimensions

- Length: 25.5mm (1.0 inch)
- Width: 18.0mm (0.7 inch)
- Thickness: standard 1.6mm PCB, plus 1.3mm components
- Weight: almost nothing

Firmware

These example VHDL firmware programs are supplied with a tif 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
  - 1200
    - flasher
    - flashctl
  - 4000
    - flasher
    - flashctl
  - common

There are separate directory trees for a tif-1200 and a tif-4000. With one exception (see the *Configuration* section) HDL code is in the *common* directory.

It could be a useful precaution to hide or rename the directory for the board you do not have. For instance, if you have a tif-1200, you could rename the 4000 directory to 4000x.

Configuration

Designs are configured for the XO2-1200HC and the XO2-4000HC FPGAs via the *tifcfg* package in *tifcfg.vhd* files in the *1200* and *4000* directories. For example:

```
-- tifcfg.vhd, 1200 version
--
-- Initial entry: 01-Jul-13 te
-- non-common definitions to personalise the tif implementations
--
--------------------------------------------------------------------------------
library ieee;                   use ieee.std_logic_1164.all;

package tifcfg is

  -- tif1200/4000 = 41h/42h = A/B
  constant TIF_ID      : std_logic_vector(7 downto 0) := x"41"; -- 'A'
  constant XO2_DENSITY : string                       := "1200L";

end package tifcfg;

--------------------------------------------------------------------------------
package body tifcfg is
end package body tifcfg;
```

Additional constants and functions can be added as a design requires. Usually the simplest practice is to define a constant in this file and use that 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="1200L" then
    return 1;
```

```
  else
    return 3;
  end if;
end;
```

Overall configuration definitions and useful constants are defined in the *defs* module *tifdefs.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.tifcfg.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 'tifcfg' files
  constant ID             : std_logic_vector(7 downto 0) := TIF_ID;
  constant DEVICE_DENSITY : string                       := XO2_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;
```

tif.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 *tiffla.vhd*.

flashctl

flashctl is more complex than *flasher* - it can be controlled from a connected PC.

As before *tiffla.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 cp2112 microprocessor on a tif board converts a bidirectional USB HID stream to a bidirectional i2c stream. The i2c stream 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
- 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 dowloaded 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 *tifwb.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 *tifdefs.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;
```

tifctl.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 tif board:

1. the application sends the data to *hidlib*
2. *hidlib* sends the data over USB to the cp2112
3. the cp2112 executes an i2c write to send the data over i2c to the FPGA's EFB
4. the FPGA state machine detects *data available* on the Wishbone interface, reads in the data and generates a write strobe
5. *tifctl.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 tif board:

1. the application sends a read request to *hidlib*
2. *hidlib* sends the request over USB to the cp2112
3. the cp2112 executes an i2c read of the FPGA's EFB
4. 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

5. the cp2112 picks up the data from i2c and sends it in a HID packet to the PC
6. the application reads the data from *hidlib*

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 tif FPGA, but the documentation can be confusing. The official route is via the *ispUFW* and *ispVM* system.

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

Software

The software supplied with a tif board supports

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

Low level functions are supplied as C/C++ programs, high level functions are in Python.

Software Installation

The software can be downloaded from http://www.bugblat.com/products/tif/tif.zip

Alternatively you can download a Git repo: https://github.com/bugblat/tif

Linux

Change to the tif/software directory and run the *setup.sh* script to install development tools:

```
apt-get update
sudo ./setup.sh
```

then run the *build.sh* script to build and install the *hdusb* and *tif* libraries:

```
sudo ./build.sh
```

Directory Structure

- hidapi
- pyhidapi
- src
    - hidapi
    - linux
    - libtif
    - windows
- static
- templates

HID access

We use Alan Ott's deservedly popular HIDAPI package (http://www.signal11.us/oss/hidapi/). HIDAPI is written in a very portable dialect of pure C.

For HID access from Python we use Austin Morton's pyhidapi package as a wrapper round HIDAPI (https://github.com/Juvenal1228/pyhidapi).

C/C++ shared library

To control your tif board you need to

- access the USB HID layer
- control the tif's onboard microprocessor (CP2112) via HID
- control the tif's FPGA via the CP2112

To ease this task we provide shared library - libtif.dll on Windows, libtif.so on Linux. libtif 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 tifwrap.h file. If you want to make intensive use of the CP2112 functions, you will need to read Silicon Labs application note AN495, available from the SiLabs web site.

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

Python Programs

All the Python programs are provided as uncompiled files. They were developed in Windows, using Python version 2.7 and the PyScripter IDE. And a health warning: they were pretty much my first contact with Python so do not use them for guidance on Python programming style.

tiffind.py

This program scans the USB bus. Here is the output from a run on my computer. The HID device in the middle three lines is a tif board, with its onboard CP2112 controller.:

```
====================hello==========================
Manufacturer:Microsoft
  Product:Microsoft 5-Button Mouse with IntelliEye(TM)
  VID:045E  PID:0039  no Serial Number
Manufacturer:Silicon Laboratories
  Product:CP2112 HID USB-to-SMBus Bridge
  VID:10C4  PID:EA90  Serial Number:u'001B3DC8'
Manufacturer:DELL
  Product:DELL USB Keyboard
  VID:413C  PID:2005  no Serial Number
==================== bye ==========================
```

tifload.py

This program takes a configuration file as input. It then

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

For example, with this command line:

```
python tifload.py tif_flasher.jed
```

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

```
====================hello==========================
Configuration file is tif_flasher.jed
Using tif library version: 'tif_lib,Jul  8 2013,19:24:28'

Manufacturer:Microsoft
  Product:Microsoft 5-Button Mouse with IntelliEye(TM)
  VID:045E  PID:0039  no Serial Number
Manufacturer:Silicon Laboratories
  Product:CP2112 HID USB-to-SMBus Bridge
  VID:10C4  PID:EA90  Serial Number:u'001B3DC8'
Manufacturer:DELL
  Product:DELL USB Keyboard
  VID:413C  PID:2005  no Serial Number
```

```
XO2 Device ID: 012ba043  - device is an XO2-1200HC
XO2 Trace ID : 00.44.30.62_62.04.80.4E
XO2 usercode from Flash:  00.00.00.00
XO2 usercode from SRAM :  54.49.46.30
reading configuration file  . . . . . . . . . . . . . 45680 bytes
erasing configuration flash ...  erased
programming configuration flash ...  . . . . . .  programmed
transferring ...
configuration finished.
=================== bye ========================
```

tifweb.py

This program implements browser control of the flashctl configuration in a tif board. You need to have installed Aaron Swartz' web.py script - see http://webpy.org/

There are several parts:

- *tifweb.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 tif 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

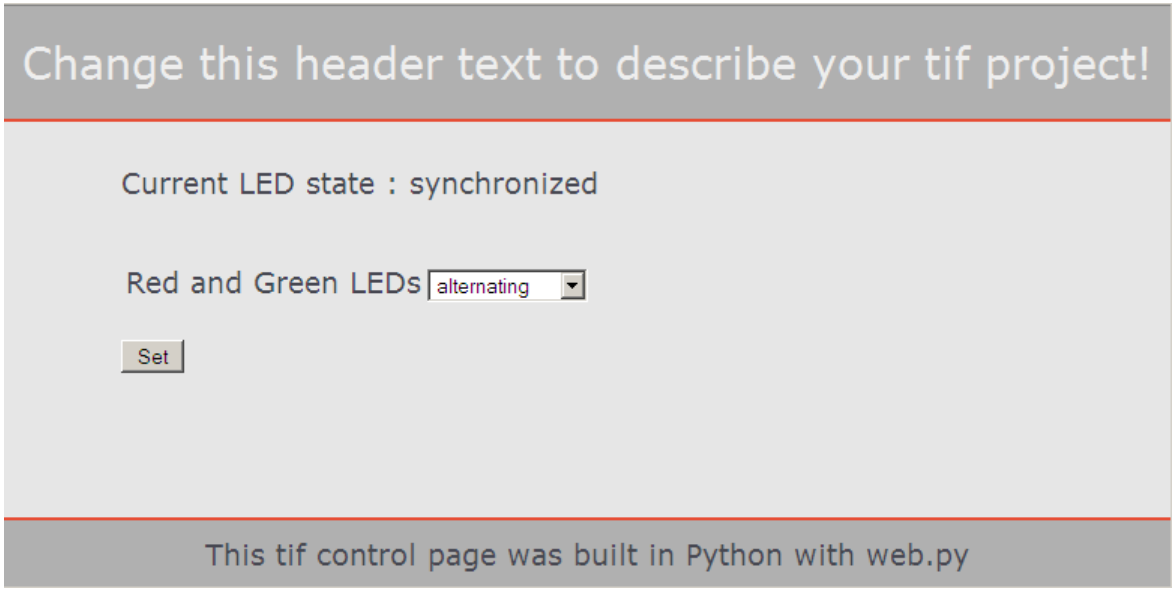
Make sure you have loaded the *flashctl* configuration in your tif board, for example via this command line:

```
python tifload.py tif_flashctl.jed
```

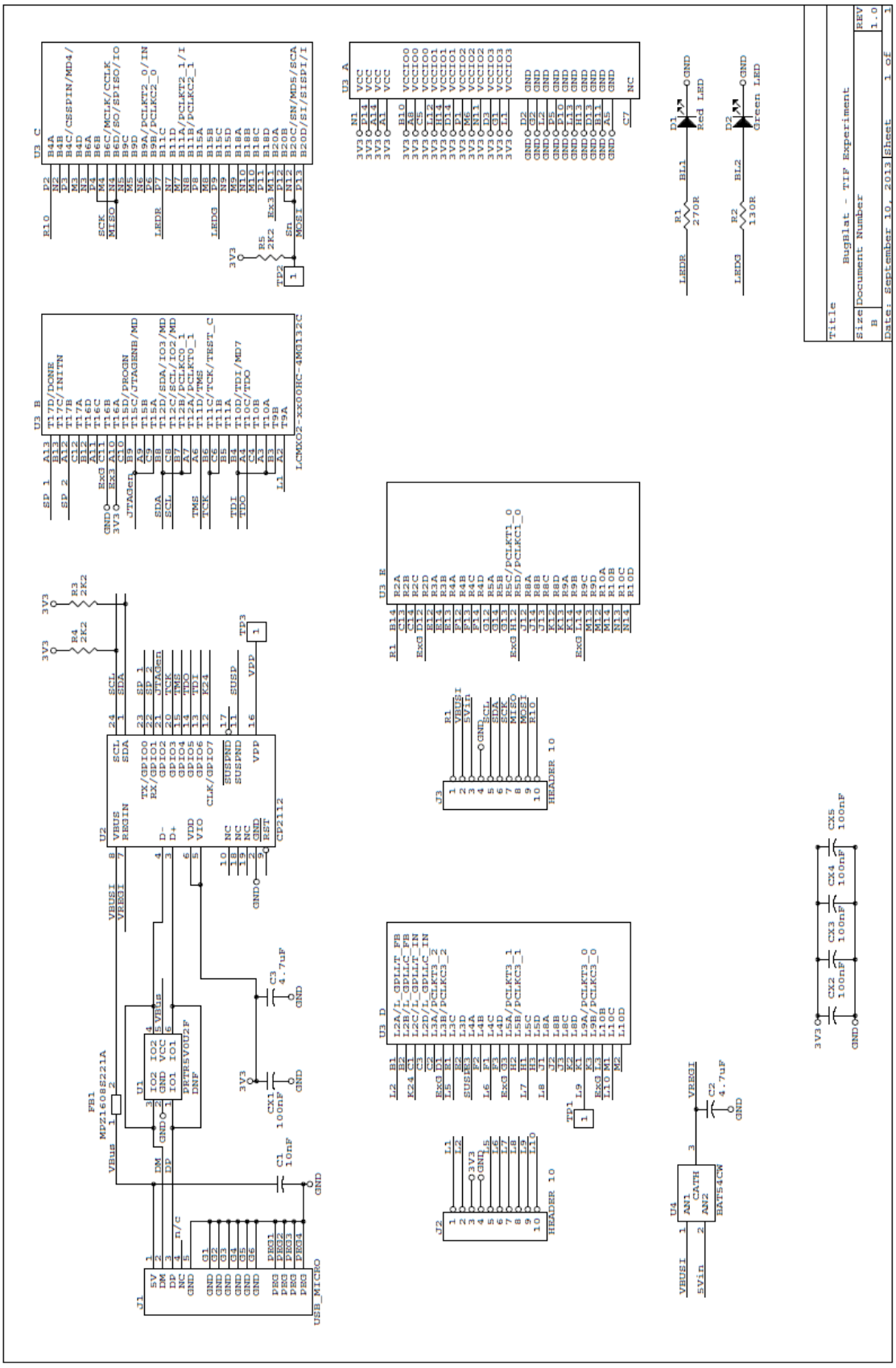
Start the program in a command window:

```
python tifweb.py
```

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



Schematic



Note that several wires, including power and ground nets, have redundant connections to the FPGA. This is a byproduct of the extremely tight (0.5mm) ball spacing on the FPGA. Redundant connections are best left unused.

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 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.