Fan Board Manual

17 October 2014

Contents

1 Quick Start **4**

1.1 Install the Drivers . . . 4

1.2 Plug In the Fan board . . . 4

1.3 More on Windows Drivers . . . 5

1.4 Software Confidence Test . . . 5

2 Hardware **6**

2.1 Block Diagram . . . 6

2.2 Functional Description . . . 6

2.2.1 Power . . . 7

2.3 Connectors . . . 7

2.3.1 J2 - 9-pin single-row Logic Analyzer Inputs . . . 7

2.3.2 JJ1 - Trigger Out Loop . . . 7

2.3.3 Test Points . . . 7

2.4 Dimensions . . . 9

3 Firmware **10**

3.1 Directory Structure . . . 10

3.2 flasher . . . 11

3.3 flashctl . . . 11

3.4 la125 . . . 12

3.5 Simulating . . . 13

3.6 Compiling and Loading . . . 13

3.7 Python Programs . . . 13

3.7.1 fanload.py . . . 13

3.7.2 fanjed2c.py . . . 14

4 Software **15**

4.1 Source Code Software Installation . . . 15

4.2 Directory Structure . . . 15

4.3 Ultra Quick Start - for Android . . . 16

4.4 Quick Start . . . 16

4.4.1 On a Windows PC . . . 16

4.4.2 On an Android System . . . 17

4.4.3 Logic Analyzer on Windows . . . 18

4.5 PC Software . . . 18

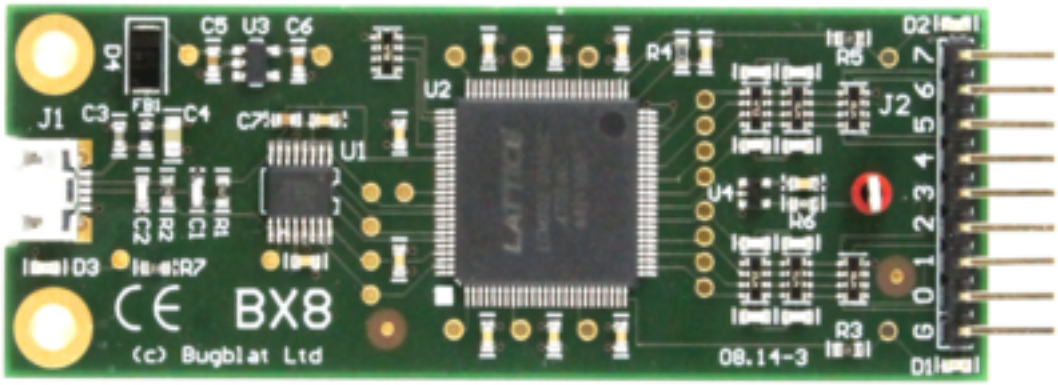
4.5.1 libbx - Shared Library . . . 18

4.5.2 bxPlugin - Gideros Plugin . . . 19

4.6 Android Software . . . 19
4.6.1 BXIF.java . . . 20
4.6.2 fanLua . . . 20
4.6.2.1 player . . . 20
4.6.2.2 slider . . . 20
4.6.2.3 me . . . 20
4.6.3 fanJava . . . 21
4.7 Lua Software . . . 21
5 Legal Stuff **22**
5.1 Regulatory and Safety Compliance . . . 22
5.2 The Design . . . 22
5.3 Fan Board Note . . . 22

1 Quick Start

This is the documentation for the Bugblat Fan board, a naked board for development and experimentation:



The Fan board connects to USB via an FTDI interface chip. To get started you need to install the FTDI drivers, then plug in the Fan board.

1.1 Install the Drivers

The Fan board software uses 100% standard FTDI *D2XX* drivers. Most Linux systems come with FTDI drivers in place, but for Windows and Mac you have to install them.

At the time of writing, the FTDI drivers are at version 2.10.00 and the driver installation program is *CDM_v2.10.00_WHQL_Certified.exe* You can search for the driver installation program via:

```
D2XX Direct Drivers site:ftdichip.com
```

Run the *driver installer*.

1.2 Plug In the Fan board

The Fan board comes with firmware already installed. when idle, this firmware flashes the onboard LEDs in phase.

So plug your Fan 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.

1.3 More on Windows Drivers

The procedure described above should have installed the drivers you need. If you had a problem, here is what you can do.

On Windows 7 and later, connect the Fan board to a spare USB port on your PC. If there is an available Internet connection, Windows will silently connect to the Windows Update website and install the standard FTDI driver.

If this procedure fails, or if you have an earlier version of Windows such as Windows XP or Vista, search on the web as follows:

```
Installation Guide site:ftdichip.com
```

and download and follow the appropriate installation guide. The guides describe how to download and run an installer from the FTDI site.

1.4 Software Confidence Test

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

Install the software bundle (see the *software* section), then change to the fpga/utils directory and load the *flasher* configuration. The command line is:

```
python fanload.py ../flasher/syn/fan_flasher.jed
```

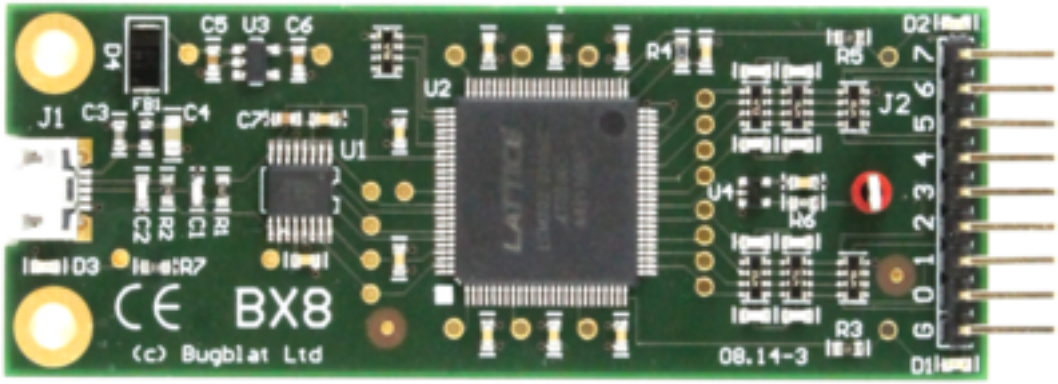
Under Linux you will probably need the sudo command prefix:

```
sudo python fanload.py ../flasher/syn/fan_flasher.jed
```

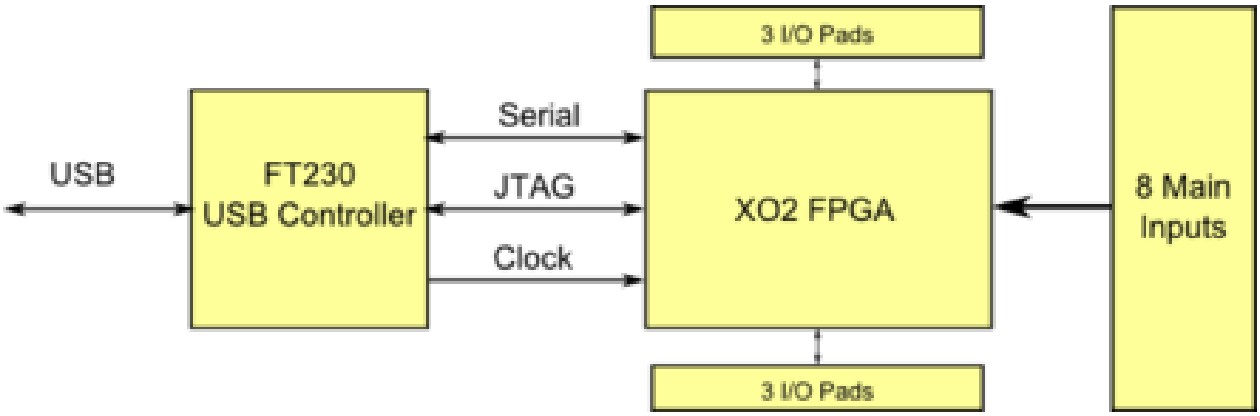
To go back to the flashctl firmware:

```
python fanload.py ../flashctl/syn/fan_flashctl.jed
```

2 Hardware



2.1 Block Diagram



2.2 Functional Description

The key components of a Bugblat *Fan* board are

- XO2-2000HC: a Lattice Semiconductor MachXO2 FPGA (details at Lattice)
- FT230X: an FTDI USB/UART interface (details at FTDI).

The XO2-2000HC FPGA contains 2112 four-input lookup tables (LUTs), on-chip SRAM and Flash, a flexible PLL, and a host of other features.

The FT230X is a USB to serial UART interface that drives the XO2's control pins. The FT230 has four UART pins and four CBUS pins. The CBUS pins are very programmable, the Fan board allocates them as follows:

Pin	Function
CBUS_0	FPGA application firmware reset
CBUS_1	24MHz clock for the FPGA
CBUS_2	FPGA JTAG Enable (JTAGENA)
CBUS_3	USB Suspend signal

The JTAGENA signal on CBUS_2 is key to the operation of the Fan board. It enables and disables the JTAG pins on the FPGA.

The FT230 UART pins are connected like this:

FT230 Pin	FPGA Application Usage	FPGA JTAG Usage
TxD	RxD	TMS
RxD	TxD	TDI
RTSn	CTSn	TDO
CTSn	RTSn	TCK

When the JTAGENA signal is Lo, the FPGA pins have their *Application* usage, when JTAGENA is Hi, the FPGA pins are dedicated to JTAG and the FPGA can be erased and reconfigured.

In normal usage the Fan board runs the FT230 UART at 3Mbit/s, easily handled in the FPGA because it is synchronous to the 24Mbit/s clock on CBUS_1.

2.2.1 Power

The XO2 runs at 3.3V, provided by an on-board regulator connected to the USB 5V pins.

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**.

2.3 Connectors

2.3.1 J2 - 9-pin single-row Logic Analyzer Inputs

Pin 1, indicated by a square pad, is a ground reference.

Pins 2 to 9 are the Logic Analyzer inputs, connected to the FPGA inputs via a protection circuit as shown on the schematic.

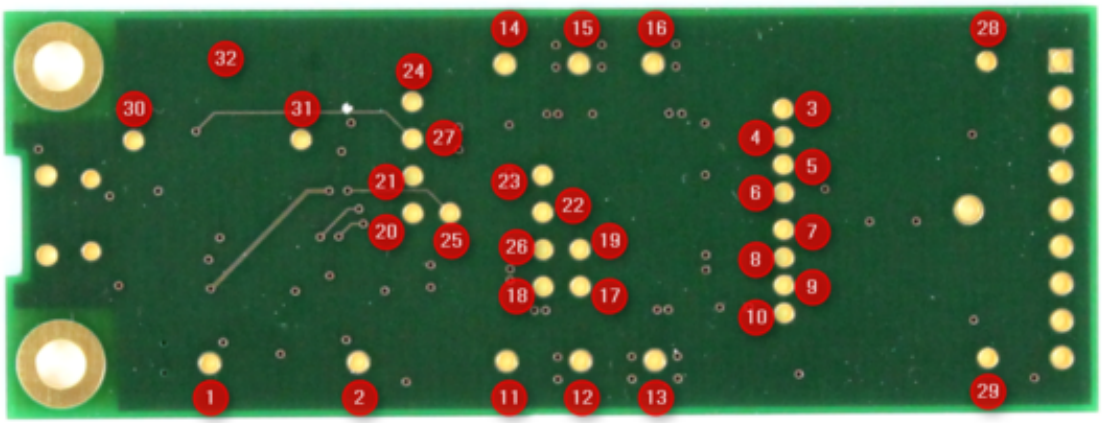
2.3.2 JJ1 - Trigger Out Loop

The *trigger out* loop is driven by a 74LVC04 inverting buffer, via a 200R series resistor. It can be used for any function programmed into the FPGA - the logic analyzer firmware uses the loop as a *trigger out* and *test clock* connector.

2.3.3 Test Points

There are 32 test points on a Fan board, as shown on this image of the rear side:

2.3 Connectors



Index	Usage
1	5V from USB
2	3.3V from regulator
3	logic analyzer input bit 0
4	logic analyzer input bit 1
5	logic analyzer input bit 2
6	logic analyzer input bit 3
7	logic analyzer input bit 4
8	logic analyzer input bit 5
9	logic analyzer input bit 6
10	logic analyzer input bit 7
11	FPGA I/O pad 75
12	FPGA I/O pad 68
13	FPGA I/O pad 58
14	FPGA I/O pad 1
15	FPGA I/O pad 8
16	FPGA I/O pad 18
17	FPGA DONE pad
18	FPGA INITn pad
19	FPGA PROGn pad
20	FT230 TxD pad
21	FT230 RxD pad
22	FT230 RTSn pad
23	FT230 CTSn pad

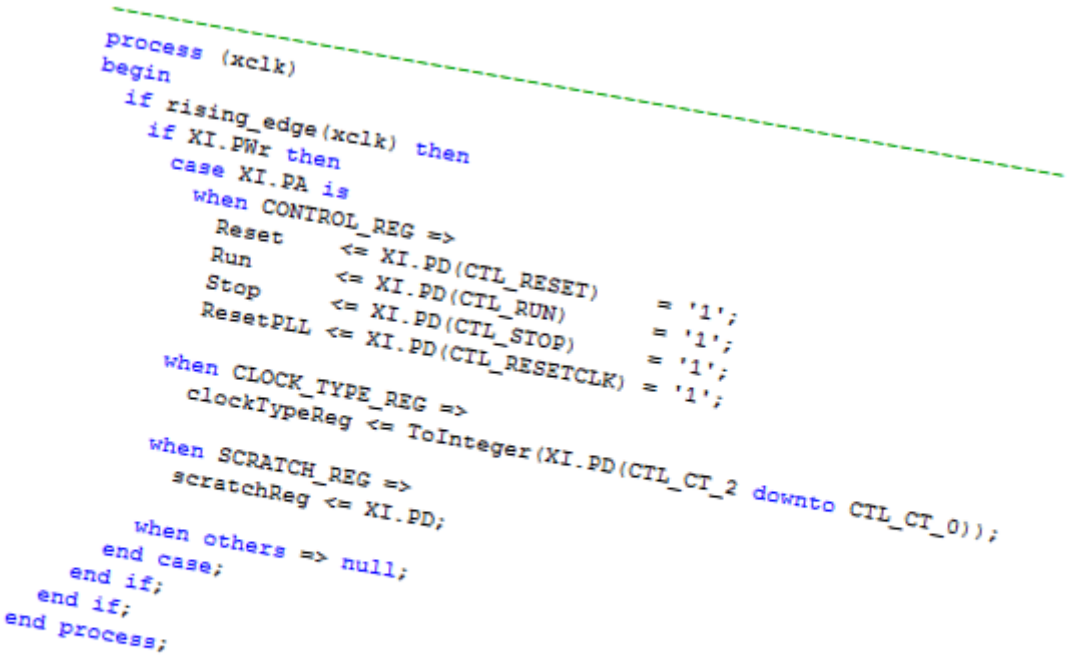
24	FT230 CBUS_0 pad
25	FT230 CBUS_1 pad
26	FT230 CBUS_2 pad
27	FT230 CBUS_3 pad
28	LED D1 anode
29	LED D2 anode
30	LED D3 anode
31	FT230 VIO pad
32	Ground

TP3 to TP10 are on a 1 mm grid. The other test points are on a 0.1 inch grid. See the schematic for further details.

2.4 Dimensions

- Length: 75 mm (2.9 inch)
- Width: 28 mm (1.1 inch)
- Thickness: standard 1.6mm PCB, plus 1.3mm components, the USB connector, the Trigger Out loop, and the Logic Analyzer probe connector
- Weight: almost nothing

3 Firmware



```
process (xclk)
begin
  if rising_edge(xclk) then
    if XI.PWr then
      case XI.PA is
        when CONTROL_REG =>
          Reset     <= XI.PD(CTL_RESET)     = '1';
          Run       <= XI.PD(CTL_RUN)       = '1';
          Stop      <= XI.PD(CTL_STOP)      = '1';
          ResetPLL  <= XI.PD(CTL_RESETCLK)  = '1';
        when CLOCK_TYPE_REG =>
          clockTypeReg <= ToInteger(XI.PD(CTL_CT_2 downto CTL_CT_0));
        when SCRATCH_REG =>
          scratchReg <= XI.PD;
        when others => null;
      end case;
    end if;
  end if;
end process;
```

These example VHDL firmware programs are supplied both in source form (VHDL) and as compiled JEDEC files:

1. *flasher* is a simple program that alternately flashes the red and green LEDs.
2. *flashctl* also flashes the LEDs, but in this case the flash pattern can be controlled over USB by an external computer.
3. *la125* is a 125MHz Logic Analyzer.

3.1 Directory Structure

- fpga
  - flasher
  - flashctl
  - la125
  - common
  - utils

Overall configuration definitions and useful constants are defined in the *defs* module *fandefs.vhd* in the *common* directory.

fan.lpf in the *common* directory is shared by all designs. It defines the pinout of the FPGA.

3.2 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 2.08 MHz.

flasher.vhd is a wrapper, the main work is done in *simpleFlasher.vhd* and *downCounter.vhd*.

3.3 flashctl

flashctl is more complex than *flasher* - it can be controlled over USB.

As before, *simpleFlasher.vhd* generates antiphase LED pulses. However the pulse stream fed to the FPGA I/Os is controlled by registers that can be written or read via the FT230 serial bus.

This is how it works. The serial stream from the FT230 is wired up UART receive (Rx) and transmit (Tx) VHDL code. The serial stream runs at 3Mbit/s, not very fast for an FPGA but the maximum speed of the FT230. A state machine in *fanuctl.vhd* handles the Rx and Tx blocks and generates a minimal internal address and data bus.

Here is the definition of the incoming address and data bus, extracted from *fandefs.vhd*:

```
type XIrec is record
  PWr         : boolean;      -- registered single-clock write strobe
  PRWA        : TXA;          -- registered incoming addr bus
  PRdFinished : boolean;      -- registered flag
  PRdSubA     : TXSubA;       -- read sub-address
  PWrSubA     : TXSubA;       -- write sub-address
  PD          : TwrData;      -- registered incoming data bus
end record XIrec;
```

Small procedures in *flashctl.vhd* listen to this bus, write values into registers, and read values from registers.

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

1. the application, typically running on a PC, executes a USB write to send the data over USB to the FT230
2. the FT230 serialises the data and sends it to the UART Rx routine in the FPGA
3. the FPGA state machine in *fanuctl.vhd* detects *data available*, registers the data and generates a write strobe
4. *flashctl.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, a D byte, or a T byte.

- If an A byte, the data field is loaded into an address register, with the six bit field allowing up to 64 addresses.
- If a D byte, the six bit data field and a write strobe go out over the internal data bus.
- If a T byte, the six bit data field is a read count, and the control state machine will read that number of bytes from internal registers and blast them out over the UART interface to the FT230

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 Fan board:

1. the application sends a trigger (T) byte to the FPGA's UART interface
2. the FPGA state machine detects the trigger and pumps out data to the FT230. It also generates a *read finished* internal strobe(s), and increments the read subaddress
3. the application executes a USB read to collect the data.

3.4 la125

la125 is a large firmware program, with many VHDL source files.

The first group sits between the USB interface and the internal control bus (the X bus):

- fantop.vhd - top level!
- fanclk.vhd - clock generator. Uses the FPGA's built-in PLL. To understand it completely you will need to read the Lattice documentation.
- fanuctl.vhd - USB interface. Takes the serial (UART) stream from the FTDI USB interface chip and generates an internal bus as described above under *flashctl*.
- fanwb.vhd - Wishbone interface. The FPGA has many built-in hard coded features which can be controlled via registers on its internal Wishbone bus. This module is the interface to the Wishbone bus.
- fanefb.vhd - Embedded Function Block (EFB) instantiation. This is where the signals in the Wishbone interface connect to the hard coded logic.

One level down are modules that sit directly on the X bus:

- fanxlayer.vhd - wires up the modules at this level.
- fanctl.vhd - control registers. The main responsibility is the signals (Run, Stop and so on) that control the logic analyzer functionality.
- fansm.vhd - central state machine. Listens to the incoming sampled data and controls the data path.

The state machine has some slave modules for specific tasks:

- fantrigo.vhd - generates the Trigger Out pulse or waveform.
- fanmatch.vhd - continuously compares the incoming sampled data with the trigger patterns being searched for.

Running across the state machine is the *data path* from incoming signal sampling to eventual storage in an on-chip block RAM. Sampling is handled in the state machine module, the other data path modules are:

- fandata.vhd - data path. Controlled by the state machine, it handles run length encoding and passes the results down to the block RAMs.
- fanbram.vhd - block RAMs. Will be instantiated once for each block RAM in the FPGA.
- fanact.vhd - activity monitor. Can be interrogated for activity on each incoming pin.

Finally we have the utility modules:

- utils.vhd
- fandefs.vhd
- downCounter.vhd

3.5 Simulating

Most of the design time with HDLs is spent in a simulator. *flashctl_tb.vhd* in the *flashctl* directory is a simulation testbed.

3.6 Compiling and Loading

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

Since a Fan board is a single chip system, we can use a simple solution - the Lattice Diamond JEDEC can be loaded directly into a Fan FPGA via the *fanload* script in the *utils* directory.

3.7 Python Programs

All the Python programs are provided as uncompiled files. We use the ctypes package for the interface between shared libraries/DLLs and Python scripts.

3.7.1 fanload.py

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

1. searches for a Fan board
2. checks whether the configuration has changed
3. clears the Fan's FPGA flash memory
4. loads the new configuration data into the flash memory
5. reinitializes the FPGA.

For example, with this command line:

```
python fanload.py fan_flasher.jed
```

this is the output from a run on my computer (the line starting *programming* has been shortened and the output may vary as program development continues):

```
====================hello=========================
Configuration file is fan_flasher.jed
Using library version: 'libbx,Aug 29 2014,12:37:59'

XO2 Device ID: 012bb043  - device is an XO2-2000HC
XO2 Trace ID : 30.1E.18.94_90.31.44.00
JEDEC file is fan_flasher.jed
starting to read JEDEC file
first configuration data line: 19
. .
last configuration data line: 570
552 frames
finished reading JEDEC file
initializing and erasing configuration flash ...  erased
programming configuration flash ...  . . . . . . . . . . . . . . . programmed
transferring ...
configuration finished.
```

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

3.7.2 fanjed2c.py

This program takes a configuration JEDEC file as input and converts it to a file that can be included in a C or C++ program.

For example, with this command line:

```
python fanjed2c.py fan_la125.jed
```

this is the output from a run on my computer:

```
====================hello==========================
JEDEC file : fan_la125.jed
Output file: fan_la125.txt
starting to read JEDEC file
first configuration data line: 36
. . . . .
last configuration data line: 1320
1285 frames
finished reading JEDEC file

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

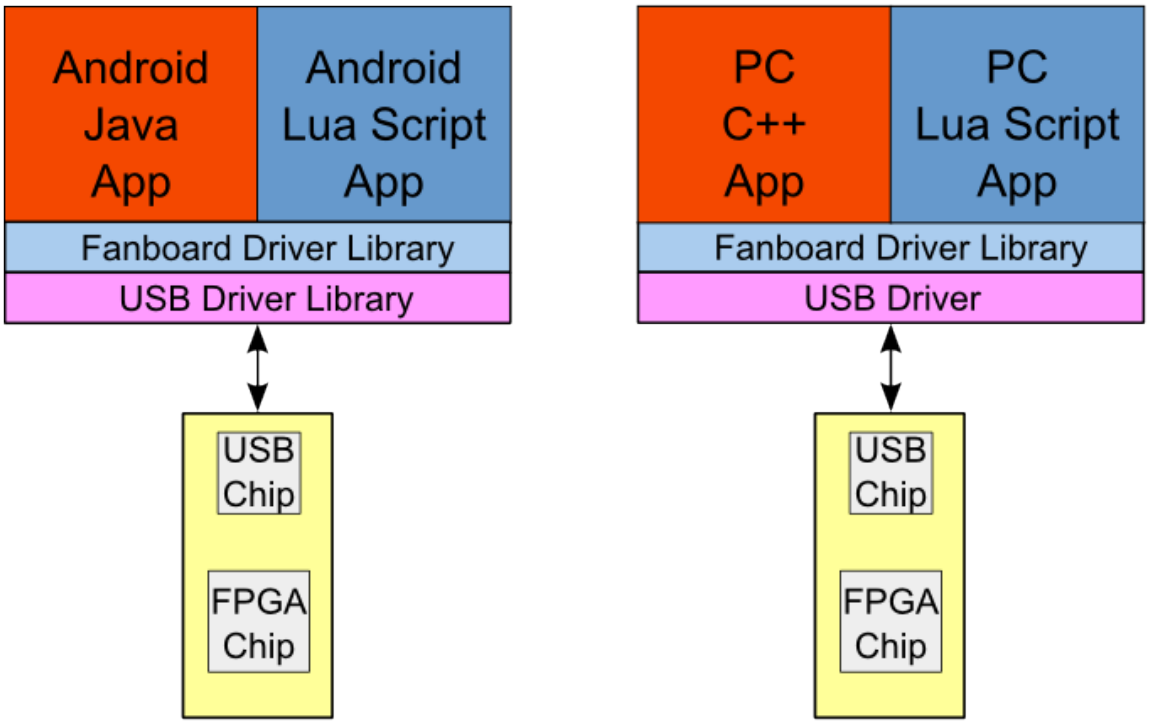
Here is part of the output file:

```
// JEDEC data from fan_la125.jed
// reversed for JTAG programming, 128 bits (one frame) per line
...
// NOTE PINS GreenLed : 52 : out*
// NOTE PINS RedLed : 25 : out*
// NOTE PINS trigOut : 36 : out*
...
0xcdbdffff, 0x00dcffff, 0x00400000, 0x09110000,
0x1214a090, 0x0062ff0a, 0x071d0000, 0x00002580,
         0,          0,          0,          0,
         0, 0x00880000, 0x10000000, 0x11000881,
0x00044088, 0x20000000, 0x00000002, 0x48000000,
         0, 0x00048090, 0x02404809,          0,
```

You include the output in a C/C++ program like this:

```
uint32_t data[] = {
#include "fan_la125.txt"
  0};
const int NUM_FRAMES = sizeof(data)/( 4 * sizeof(data[0]) );
```

4 Software



The software supplied with a Fan board supports

- loading a configuration into the Fan board
- controlling the Fan board from a PC
- controlling the Fan board from an Android device
- controlling the la125 firmware example

Low level functions are supplied as C/C++ or Java programs, high level functions are in C/C++ and Python, and examples are in Java or the Lua scripting language.

In addition, we supply a GUI/control program for the *la125* firmware as a Windows executable.

4.1 Source Code Software Installation

The source code software can be downloaded from http://www.bugblat.com/products/fan/fansw.zip.

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

4.2 Directory Structure

- pc
  - libbx
  - bxPlugin
- android
  - fanLua

    - fanJava
- lua
    - leds
    - me

The python utilities are in the *../fpga/utils* folder.

4.3 Ultra Quick Start - for Android

For an expert, possibly with Android SDK/NDK experience.

Hardcore:

- install *fan-me.apk* on your Android and start the app
- edit the code in the sdcard/gideros/me/resource/main.lua folder

Slightly less hardcore:

- install *fan-player.apk* on your Android and start the app
- install Gideros and/or ZeroBrane on your PC
- run or edit and debug the lua/leds code

4.4 Quick Start

There's a lot of software here. This section runs through the minimum steps needed to get something working on your system. It assumes that you have

- carried out the basic installation of FTDI drivers as described on the *start* page
- downloaded the Fan software bundle

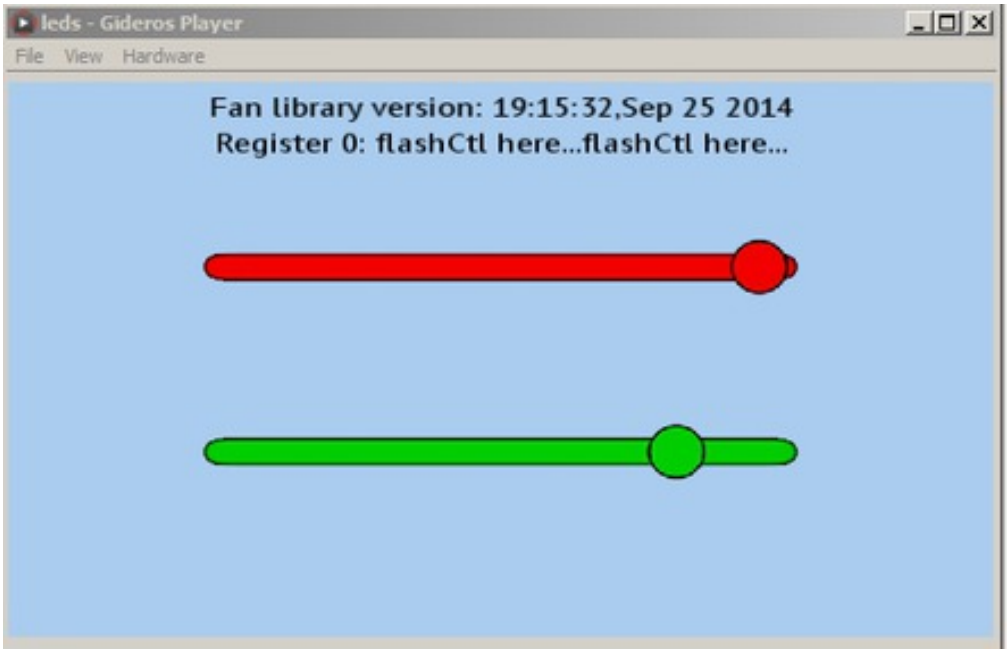
4.4.1 On a Windows PC

Start by using your PC to install the *flashctl* firmware on the Fan board. A factory fresh board will already have this firmware installed:

```
python fanload.py fan_flashctl.jed
```

Then:

- download and install Gideros
- copy *bxPlugin.dll* to the Gideros Plugins folder
- start Gideros and run the *lua/leds* example You should be able to control the LEDs on the Fan board, as below



4.4.2 On an Android System

Start by using your PC to install the *flashctl* firmware on the Fan board. A factory fresh board will already have this firmware installed:

```
python fanload.py fan_flashctl.jed
```

The next stage is to install Gideros Studio on your PC and Gideros Player on your Android:

- download and install Gideros
- install the Gideros Player (GiderosAndroidPlayer.apk in the Gideros install folder) on your Android. Search on the net for instructions as to how to do this. Installing via Dropbox is often the easiest solution
- start Gideros Studio and load any example application
- start the Gideros Player on your Android. That will display the IP address of the Android - you need to type this into *Player Settings* in Gideros so that Gideros Studio on your PC can find Gideros Player on your Android
- start the app within Gideros Studio. It should run on the Android

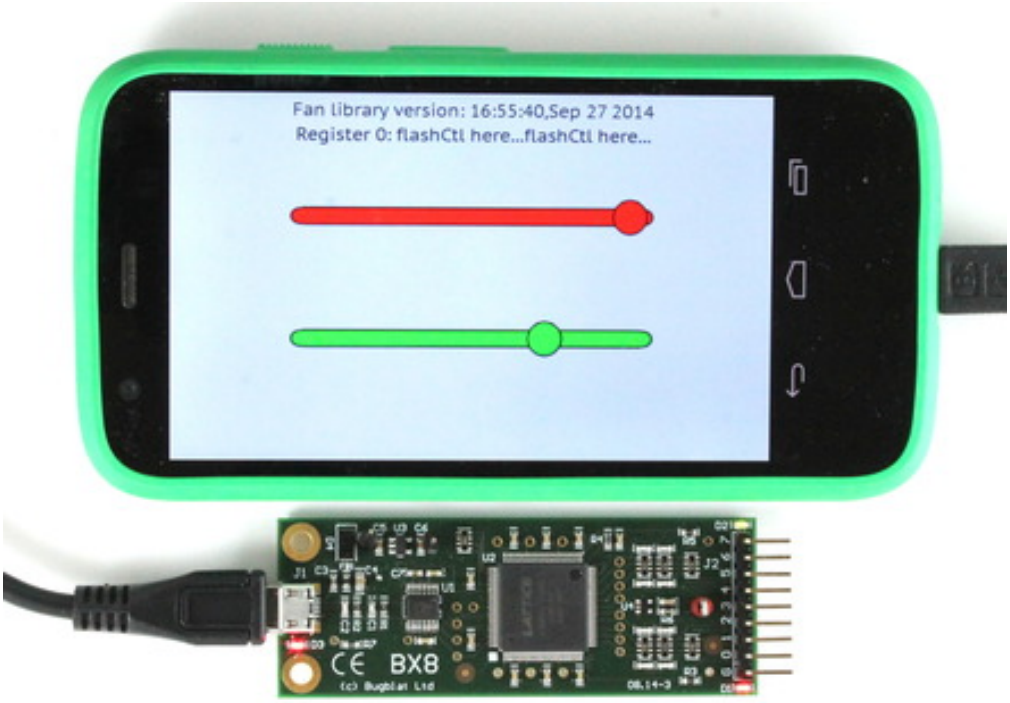
At this stage we have verified that Gideros Player on your Android is communicating with your PC.

To talk to the Fan board we need to run a custom version of the Player. Since Android apps potentially run forever, we may need to stop the Gideros Player. There's quite a good chance that the Player has terminated, but if not you can stop it via the Settings/Apps/Running widget. Next:

- install *fan-player.apk* from the *output* folder of fanLua. This bundle may be called *fan-player-debug.apk* or *fan-player-release.apk*
- start the *Fan Player* app on your Android
- check that a Gideros app from your PC will run on your Android
- plug your Fan board into your Android. The Android takes an OTG cable - most OTG cables are very short and are terminated with a socket which will accept a USB memory stick. A regular USB

cable, usually a little longer, plugs into the Fan board. Then join up the two cables and the LEDs on the Fan board should start flashing

- In Gideros Studio, start the *lua/leds* application. You should be able to control the LEDs on the Fan board, as below



4.4.3 Logic Analyzer on Windows

Start by using your PC to install the *la125* firmware on the Fan board:

```
python fanload.py fan_la125.jed
```

Then download and run the installer. The Logic Analyzer software should be installed and ready to run.

4.5 PC Software

4.5.1 libbx - Shared Library

To control your Fan board you need to

- access the USB port and control the onboard FT230
- reload the FPGA
- control the FPGA

To ease this task we provide a shared driver library that sits directly above the FTDI chip drivers. The *libbx* library operates at the level of *write to the FPGA*, *read from the FPGA*, and *reload the FPGA*. It knows nothing about the firmware inside the FPGA, nothing about the register allocations.

libbx operates in two modes.

- in application mode, libbx interfaces between an application program and the operating system drivers for the FT230 USB/UART chip.
- in JTAG mode, libbx controls the FT230 so that it can bit-bang the JTAG interface; it also implements the host of low-level commands that are required to reprogram the FPGA.

libbx 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 libbx folder. The interface is defined in the *libbx.h* file.

The software/libbx folder includes precompiled libbx.dll and libbx.so files - you can run the Python software even if the compiler tools for C/C++ are missing from your system.

4.5.2 bxPlugin - Gideros Plugin

The Fan board supports the Gideros software framework, recently released as open source. *bxPlugin* is a thin layer over libbx that bridges between a Lua program in Gideros and the Fan hardware.

The plugin compiles under Windows to a DLL. To install it under Gideros the DLL must be copied to the *Plugins* folder of your Gideros installation. Usually this is here:

```
C:\Program Files\Gideros\Plugins
```

You may need to run as Administrator.

With the plugin installed, it can be used in a Lua script like this:

```
require "bxPlugin"

if bx.open() then
  isOpen = true
  local ok, d = bx.readReg(0, 32)
  ...
  ...
```

See also the Lua examples.

4.6 Android Software

We provide

- BXIF.java - a Java driver for the Fan board
- fanLua - an example set of Android Studio java/c++ programs for running Gideros Lua scripts on the Fan board
- fanJava - an example Android Studio java program which accesses a Fan board

For best use of our Java software you need to download and install the Android Studio/SDK package. We don't support the Android Eclipse/ADT software environment. For Lua and other languages such as Python you also need the Android NDK package. There are numerous online references to downloading and installing these packages.

A typical Android device has a single USB connector, so for debugging a USB OTG application you need to establish a wifi connection between your PC and your Android device. There are numerous online guides to this as well.

4.6.1 BXIF.java

BXIF.java is the Java language equivalent to libbx on a PC. Whereas libbx sits above FTDI's D2XX driver library, BXIF.java sits above FTDI's j2xx.jar driver package.

As far as possible, BXIF implements the same functions as libbx, with the exception of the rather specialized functions used to reprogram a Fan board.

BXIF.java is used in both the fanLua examples and the fanJava example. It sits in the source tree for fanLua here:

```
fanLua/app/src/main/java/com/bugblat/bx
```

4.6.2 fanLua

fanLua implements the Gideros Lua framework. It is an Android *flavors* application, with three flavors. The three flavors can be seen in the source tree under src. src/main holds the core application. src/player holds the parts that are different for the player flavor, similarly for src/slider and src/me.

It is possible to extend the core application, for instance to access Android functionality such as wifi, SMS, and email. In that case you will need to add to the functions in the jni parts and in BXIF.java. See README.txt in the main folder.

4.6.2.1 player

The *player* version has a completely empty *assets* folder. As a result it will act as behave exactly as the standard Gideros player, listening for incoming commands from Gideros Studio or from a debugger such as ZeroBrane.

4.6.2.2 slider

The *slider* version contains the *lua/leds* application, compiled, in its assets folder. When this app is installed and run it will search for and control an attached Fan board.

You can easily replace the default application. Search for a guide to exporting an app as *assets only* from Gideros. Copy those assets into the *assets/assets* folder and rebuild the project.

Or see the next section of this manual.

At the time of writing, you may be unable to export from the default Gideros download. Gideros is transitioning to open source and you may need to upgrade from the free version to a licensed version (also free).

4.6.2.3 me

As delivered, the *me* version of the app contains the standard *Button* application. It will run OK, but it is really a placeholder for your own application.

This is how it works.

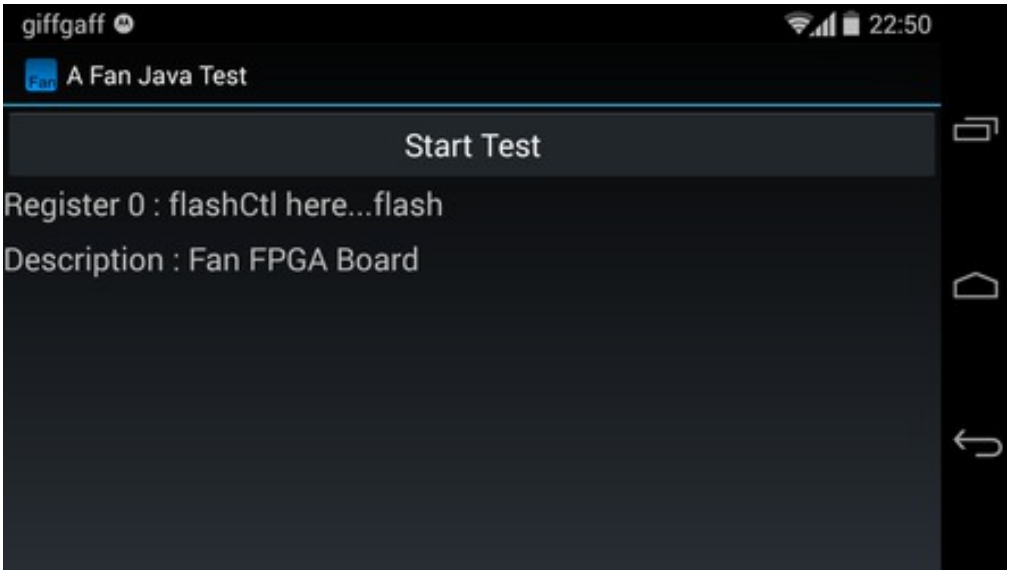
- install and run the *fan-me.apk* app.
- using your favorite file manager app on the Android (we use ASTRO), inspect the sdcard/gideros/me/resource folder. This may become sdcard0/... or something similar
- the resource folder should contain the script for the *me* app. Via a file manager, you can replace it and that's the new app!

You will probably want to develop and interactively debug the replacement script with the *fan-player* version.

4.6.3 fanJava

The fanJava app is almost the simplest possible Android Java app for the Fan board. Almost because there is only one screen, but not quite because that screen is a *Fragment* rather than a traditional *Activity*.

The app uses the BXIF library to access the Fan board. The outermost Activity is MainActivity.java, the Fan code is in the FanDemo.java Fragment class.



4.7 Lua Software

Two sample Lua programs are supplied. As described above, the lua/me program is a simple modification of the Button program in the standard Gideros distribution.

lua/leds demonstrates use of the Fan driver. The Fan board is accessed via the bxPlugin routines, so the program starts with:

```
require "bxPlugin"
```

After that, plugin functions can be called via the bx table. For instance:

```
if bx.open() then
  local ok, d = bx.readReg(0, 32)
  ...
```

The functions implemented on Android are listed under fanLua in the src/main/jni/bxPlugin.cpp file, and the driver routines they access are in BXIF.java.

The functions implemented on a PC are listed in the pc/bxPlugin.cpp file, and the driver routines they access are in libbx.dll(so).

5 Legal Stuff

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

5.1 Regulatory and Safety Compliance

The Fan board is not a complete product and 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 regulatory and safety compliance standards and related certifications as may be applicable. You will employ reasonable safeguards to ensure that your use of either product will not result in any property damage or injury or death, even if the product should fail to perform as described or expected.

5.2 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.

5.3 Fan Board Note

This product 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.