

Table of Contents

Step 2.....	1
Learn FPGA Design Flow with a Frequency Divider Circuit.....	1
Adapted from Chapter 2 of “Make: FPGAs”	1
Hardware Hacking Hello World.....	2
Extending Hello World	2
Select Your FPGA Board	3
Get an LED Blinking	4
What Did We Accomplish?	4
Next Step: Learn about Concurrency with a Digital Clock0	5
Blink Two LEDs on iCEstick	5
Blink LEDs with Red Pitaya Visual Programming	10
Setup Visual Programming	10
Programming With Visual Programming	17

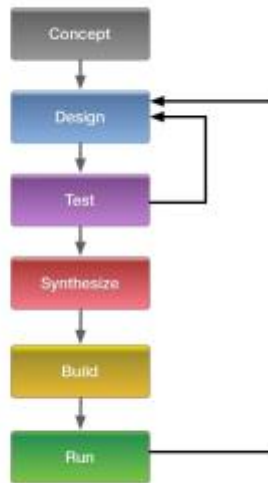
Step 2

Fair warning. You are about to enter a battle zone. This is not easy! It is fraught with peril and can be frustrating. Things worth doing are rarely easy. Some boards are easier to work with than others. Step 2 is where you sort out your particular development environment and document the steps required for someone else to duplicate it.

At all stages, we are going to share our work and help each other out. Radio functions are not mysteries. They are well-studied and can be mastered. Persistence reveals results!

Learn FPGA Design Flow with a Frequency Divider Circuit

Adapted from Chapter 2 of “Make: FPGAs”



Above is a diagram of how things like FPGA design almost always happen.

We begin with a **concept**.

We **design** our implementation.

We **test** the implementation, use what we learn to adjust the design, and then test the result.

We may do this many times!

Once we're satisfied with the result, we **synthesize**, **build**, and **run**.

When we run the completed design, we **test it again**.

It is highly likely that we may find that we need to make additional adjustments to the design, and we have to go back to the **design-and-test** loop. This is normal!

Hardware Hacking Hello World

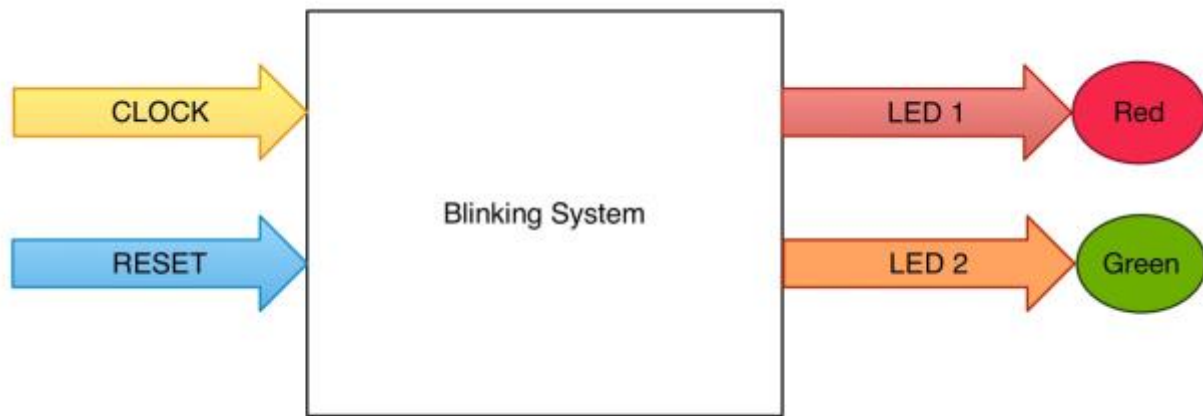
You may already be familiar with the Hello World concept, where the phrase "Hello World" is printed out on a screen or readout. In the software world, "Hello World" is often the first real test that the system can be programmed and is working.

On the hardware side, when you want to test that the system can be programmed and is working, the iconic "Hello World" is blinking an LED.

Extending Hello World

For this step, we're going to extend the basic "Hello World" of blinking an LED. We are going to combine blinking an LED with dividing down a fast signal to get a slower one. We will also have a signal that overrides the basic behavior.

Here's a block diagram of what we're doing.



Our inputs are system clock (CLOCK) and reset (RESET). When documenting work like this in drawings, inputs generally come in from the left and flow to the right. Our outputs are signals that blink two LEDs (LED 1 and LED 2). Outputs also flow from left to right. Blocks are used to abstract functions or collections of functions. High level block diagrams are the simplest and most abstracted form of documentation. Each block can be replaced with a more detailed, or lower level, block diagram.

We are going to make the LEDs blink at different rates. The output blinking rates we are going to produce with the Blinking System are much lower than the system clock. If we used the system clock directly to blink LEDs, the LEDs would blink at a rate much faster than what we could physically see. We have to divide down the system clock rate to get LED blink rates that can be seen with human eyes. Being able to control the LED behavior with a reset signal is an important concept. Signals from user interface and sensors need to be successfully incorporated into the system design all along the process. We're beginning with a very simple reset signal, but this control signal concept will be extended to numerous signals for a working radio.

Select Your FPGA Board

In order to do this step with physical hardware, **you will have to choose some sort of evaluation board or development module or testbed. You need an FPGA that can be programmed.**

For Phase 4 Ground, we are going to be using a variety of SDRs, and even make our own!

Do you have an SDR with an FPGA that Vivado can talk to? Great! If you need one, then this is your chance to go get one!

You can tell whoever you need to that Michelle said it was ok!

Vivado talks to mid-scale and up Xilinx parts like Ultrascale, Virtex-7, Kintex-7, Artix-7, and Zynq-7000 series. If you have a board with any of these parts, then Vivado is what you need. It's what you installed in Step 1.

If you want to use smaller Xilinx parts, like the Spartan-3, Spartan-6, Virtex-4, Virtex-5, and Virtex-6 families, then you will need the (discontinued) Xilinx ISE. Xilinx ISE is in "sustaining" mode, which means no new versions will be released, but you can still install it and use it.

We strongly recommend Vivado. It's the current version of the toolchain from Xilinx. The chips supported by Vivado are what we're going to be dealing with.

Different boards may need different versions of Vivado. That's ok. Install whatever is called out for your board. Don't fight it, just install it.

For example, let's look at the Red Pitaya.

<http://pavel-demin.github.io/red-pitaya-notes/led-blinker/>

These are a set of notes to get the Red Pitaya cooperating with Vivado, and also blinking an LED. Do you have a Red Pitaya? Then the link above is a great start!

If you're using something like a USRP x310, then according to the Ettus website, Xilinx Vivado 2015.2 Design Suite is what you'll need. Don't bang your head against a wall. If something blows up, back off and double-check.

Getting things set up for development can be hard. Tribal lore, unclear directions, things that change out from under you – all of this and more is considered to be part of the embedded development landscape.

This is not an excuse. Difficult or badly designed environments should not be normal or put up with without complaint. However, dealing with the innards of an FPGA is not the same as firing up a word processor and printing off a document. With great power comes great responsibility and almost always a steep learning curve. A good attitude (and sympathetic co-conspirators) is irreplaceable!

[Get an LED Blinking](#)

This section will be updated as people document their recipes.

[What Did We Accomplish?](#)

We chose a board.

We got it working with Vivado.

We blinked an LED.

We implemented an LED **Blinking System**.

- We learned how to divide down the system clock to get useful human-rate signals.
- We added a control signal.

Next Step: [Learn about Concurrency with a Digital Clock0](#)

Blink Two LEDs on iCEstick

by Carl Wall, VE3APY

The iCEstick is an iCE40 evaluation dongle by Lattice semiconductor. It is a USB stick which has a FTDI FT232RL usb to serial chip, a 32Mbit SPI flash memory and a iCE40HX1K FPGA chip with some leds and a PMOD connector. To program iCEstick the open source IceStorm project software will be used. Since the IceStorm software only uses verilog that is what will be use for this example. The link to IceStorm build information is at

<http://www.clifford.at/icestorm/>

The manual for the iCEstick can be downloaded near bottom of web page.

<http://www.latticesemi.com/icestick>

Two of the more common FPGA Hardware Description Languages are VHDL and Verilog. You can have lots of flame wars over which is best. The bottom line if you are doing a lot of FPGA programming, the company that is paying you will normally decided which to use. Here is a little video which gives a quick overview.

<https://www.youtube.com/watch?v=frBnuKeshoM>

I heard one comment that VHDL tries to prevent you from shooting yourself in the foot, but Verilog will let you do it because you know what you are doing. Since I have a book in one hand and coding with the other, I have no opinions yet. ;-)

The second request from Michelle was to blink two leds and and also have a reset to the counter for the leds. This will require that we count down the crystal frequency for the leds. Since google is our friend we will do a google search to find some examples. The two that I used are

<http://badprog.com/electronics-verilog-blinking-a-led>

<http://simplefpga.blogspot.ca/2012/06/code-to-make-led-blink.html>

I also used this code example for an example of a shell script file for the building of a project for the iCEstick using the IceStorm software.

<https://github.com/wd5gnr/icestick>

Two files will be needed, the verilog.v code file and an icestick.pcf file which is what pins are connected to the outside world and what we would like to call them.

Lets start with the verilog code.

blink2.v

```
/* *****/
/*      blink2.v          */
/*      Nov 23, 2016      */
/*      VE3APY            */
/*      Carl Wall         */
/*                      */
/* *****/

/*  module */
module blinking_two_leds (

    input clock_12, // 12 MHz clock on board
    input PMOD1,    // J2 pin 1
    output LED1,    // Red led
    output LED5     // Green led
);

/*  reg */

reg [32:0] counter;

/*  assign */

assign LED1 = counter[24]; // Red led
assign LED5 = counter[25]; // Green led will blink half rate of red one.
assign reset = PMOD1;

/*  always */

always @ (posedge clock_12 or posedge reset) begin
    if (reset)
        counter <= 0;
    else
        counter <= counter + 1;
    end

endmodule
```

It is always good form to start with a header block which gives the name of the program, the date and who to call when it does not work. These days the type of license should also be in there, for anyone who might want to reuse your code.

The next thing in this program is to tell Verilog what names will be attached to inputs and outputs. In our case we have two inputs, the 24 Mhz clock and a signal from a switch to do a reset. We have two outputs LED1 which is a red LED and LED5 which is a green LED.

The next block of code we let Verilog know we will be using a counter which is 33 bits long by use of the **reg [32:0]**. Remember that we are counting from zero so there is 33 bits not 32. We then put what we want to call this counter, OK I was not original with the name. In a larger program it is good to give some thought to your names.

The next part is being used to change the names of some of the signals, I have assigned two bits from the counter chain to LED1 and LED5, and assigned **PMOD1** signal to the label **reset** which will allow me to use reset later which is a name people will know.

The **always** is where the meat of how things will work is described. On the positive level of the reset signal the counter will be loaded with zero. Otherwise the counter will be incremented by one on the positive edge of the clock_12 pulse.

So how did I decide where to tap the signals to drive the LEDs. Since I am math challenged I looked at the examples and picked a tap which I thought would be in the ball park. Compiled the code, and downloaded it. The flashing was pretty slow, so I reduced the tap number on the counter chain, until it looked right. This approach is OK with a small program like this, but a larger program which might have a much longer compile time this approach could be very slow.

Then I checked that the reset switch did what I thought it would do and was finished. I also noticed that some of the leds which I did not use were on very dim, so the floating pins I did not use was causing that. So if I was sending this off to production I would add code to make those unused led pins low.

blink2.pcf

```
# blink2.pcf
#

#Clock
set_io clk_12 21 # 12 MHz clock

# LEDs

set_io LED1 99 # red
set_io LED5 95 # green

#PMOD
# Note: pin 5 and 11 are ground, pins 6 and 12 are VCC
set_io PMOD1 78 #PI01_02 Reset
```

blink2.pcf is the constraints file. When the compiler/router is figuring out how to build your circuit in the FPGA it will use this file to limit where it can put the input/output connections from the FPGA design to the pins on the

FPGA chip. The file is a simple list of the names you have called things and the chip pin you are expecting them on. Some versions of software do not like you naming things and not using them in your FPGA, so some constraints files can have a lot of unused names commented out. It is nice to have a common constraints file for the board that you are using, saves some trouble shooting later. A more complicated FPGA chip would also have more options in the constraints file for the various pins.

We now have a Verilog file and a constraints file, so lets build the project. We are going to start by doing it on the command line and then using a shell script to do the same thing. I am going to assume that you followed the IceStorm build instructions and have the tool chain installed.

We will open a terminal window and **cd** to the folder where we have the Verilog and constraints file.

```
yosys -p "synth_ice40 -blif blink2.blif" blink2.v
```

This command will convert the verilog file to a RTL file in the blif format. Check the web page for yosys for more details. This is black magic to me.

```
arachne-pnr -d 1k -p blink2.pcf blink2.blif -o blink2.asc
```

This command takes the blif file and the constraints file and figures out how to build the design in the FPGA chip. It produces an ascii file of the result. Or in the words of the developer. Arachne-pnr implements the place and route step of the hardware compilation process for FPGAs. It accepts as input a technology-mapped netlist in BLIF format.

```
icepack blink2.asc blink2.bin
```

Takes the ascii file and converts it into a binary file which will be loaded into the SPI flash on the iCEstick.

```
iceprog blink2.bin
```

This command takes the binary file and downloads it to the Flash on the iCEstick. Since this is over USB some trouble shooting is sometimes required. If the command fails but works if you do use, `sudo iceprog blink2.bin` then you have a permission problem when the iCEstick got mounted. Put the following information

```
ACTION=="add", ATTR{idVendor}=="0403", ATTR{idProduct}=="6010", MODE:="666"
```

into the file named 53-lattice-ftdi.rules
and put that into `/etc/udev/rules.d/`

Now that we can built the project at the command line, when you are building it a lot this can lead to a lot of typing so here is a shell script which is put in the same folder as the verilog and constraints files to automate the procedure.

build.sh

```
#!/bin/bash
# Build with open source tools
if [ -z "$1" ]
then
    echo Usage: build.sh main_name [other .v files]
    echo Example: ./build.sh demo library.v
    exit 1
fi
```



```
set -e # exit if any tool errors
MAIN=$1
shift
echo Using yosys to synthesize design
yosys -p "synth_ice40 -blif $MAIN.blif" $MAIN.v $@
echo Place and route with arachne-pnr
arachne-pnr -d 1k -p blink2.pcf $MAIN.blif -o $MAIN.txt
echo Converting ASCII output to bitstream
icepack $MAIN.txt $MAIN.bin
echo Sending bitstream to device
iceprog $MAIN.bin
```

`./built.sh blink2` is the command we would now used to build the project

I would like to thank Clifford Wolf for developing IceStorm and the other tools which lead to the Open Source development tools for the Ice40 FPGAs.

Appendix

Good overview of the steps of using the IceStorm software.

<https://www.youtube.com/watch?v=1CNVsxoLI60>

Talk by Clifford Wolf at the 32C3 about the IceStorm project

<https://www.youtube.com/watch?v=9rYiGDDUlg>

Links to the various software parts of IceStorm

<http://www.clifford.at/icestorm/>

<http://www.clifford.at/yosys/>

<https://github.com/cseed/arachne-pnr>

Books which I thumbed through while trying to code blink2.v

Verilog by Example A concise introduction for fpga design
by Blaine C. Readler

Programming FPGAs Getting started with Verilog

by Simon Monk

Make: FPGA
by David Romano

Blink LEDs with Red Pitaya Visual Programming

By Michelle W5NYV

I set up my Red Pitaya for Visual Programming before attempting to connect to it with GNU Radio and Vivado. Visual programming is a web-based approach to programming that simplifies coding down to drag-and-drop blocks. It's intended to make Red Pitaya more accessible in educational situations.

Setup Visual Programming

In order to use Red Pitaya visual programming, there is a license that you have to buy. At the time of this writing, the cost was 5 euros per month. You subscribe through the Red Pitaya store. I did this and it was fast and easy, just like all of the other Red Pitaya store experiences have been so far.

Visual Programming on the Red Pitaya requires internet access. Internet access is also required when upgrading the Red Pitaya operating system, installing applications from the marketplace, and unlocking licenses for applications from the marketplace.

Traditional coding on the Red Pitaya, like using it with GNU Radio and/or Vivado, does not require internet access.

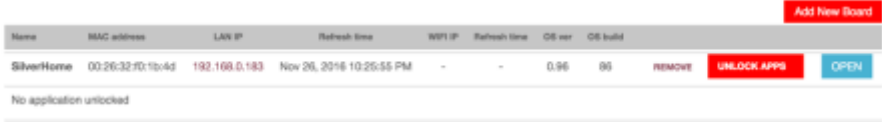
Basic setup for the Red Pitaya needs to be accomplished before setting up Visual Programming. The major tasks are unboxing, obtaining a proper power supply, updating the operating system, and being able to connect to it by using a web browser. Documentation on how to accomplish the above setup can be found at:

<http://redpitaya.readthedocs.io/en/latest/doc/quickStart/needs.html>

You will need to log in to your Red Pitaya account. You will then add your board to your account. When successful, the board should show up in the My Equipment page. For me, the link was

<https://store.redpitaya.com/myequipment/list/>

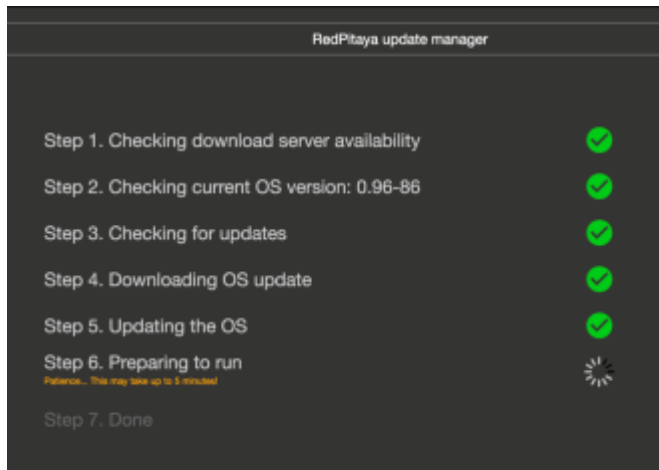
And the page (after setting things up) looked like this:



Name	MAC address	LAN IP	Refresh time	WiFi IP	Refresh time	OS ver	OS build	
SilverHome	00:26:32:70:1b:4d	192.168.0.183	Nov 26, 2016 10:25:55 PM	-	-	0.96	86	REMOVE UNLOCK APPS OPEN
No application unlocked								

Visual programming license: Valid until 2017-04-03 22:14:17. Visit Visual programming HOPR.

Here's what the task of updating the operating system (OS) looks like.



This is updating the OS after you've prepared the SD card and successfully set up the Red Pitaya. Depending on when you bought your Red Pitaya, you may have to set up the SD card yourself, or it might have the operating system already installed. Either way, regular updates to the OS are made and should be installed. The update OS function can be found by clicking the IP address under LAN IP and selecting the OS update menu option.

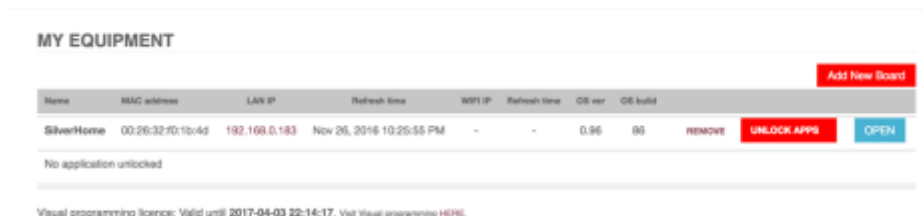
Completely setting up the Red Pitaya for Visual Programming is not hard, but there are some fiddly bits. All of the steps were accurate at the time of this writing, but things may change in the future.

Red Pitaya has a video tutorial for blinking LEDs from November 2015. You can view it here:

<https://www.youtube.com/watch?v=V4ZSB8oetDQ>

We're going to duplicate the steps, expand on a few of those steps, and show updated screenshots.

Go to the My Equipment page.

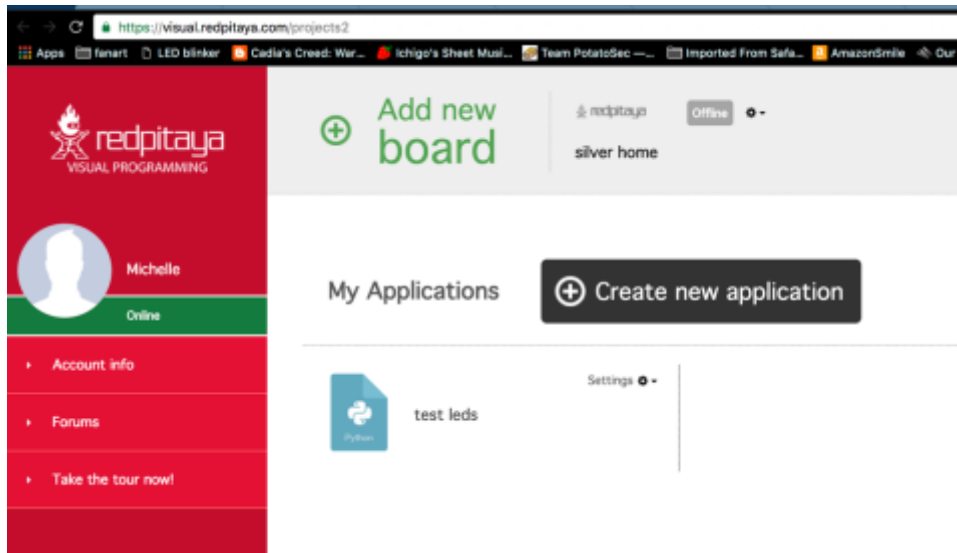


Click the link Visit Visual programming [HERE](#).

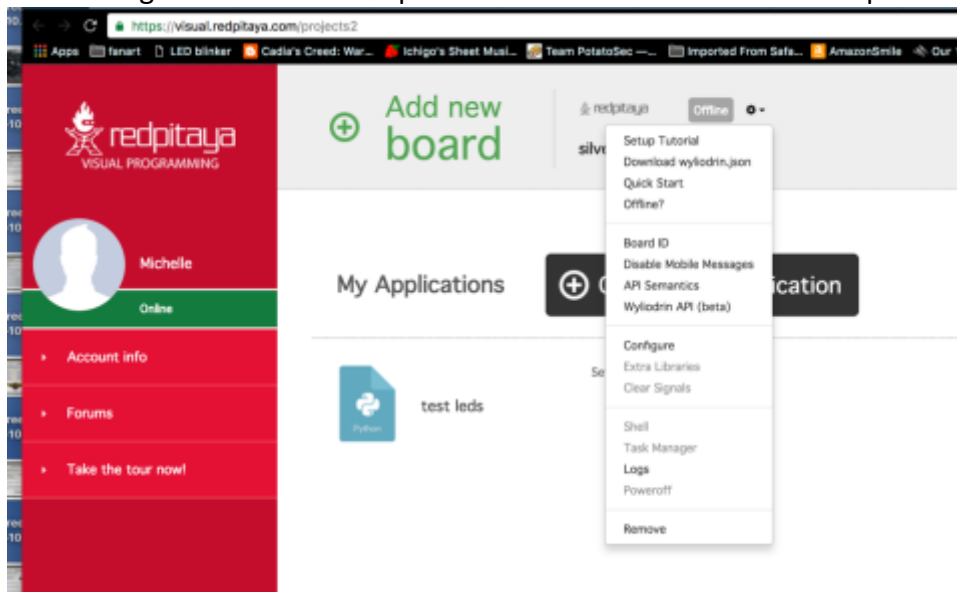
This launched the page

<https://visual.redpitaya.com/>

You should see a page that looks something like this. I needed to add my board on this page to get it to show up, even though it was successfully listed on the My Equipment page.



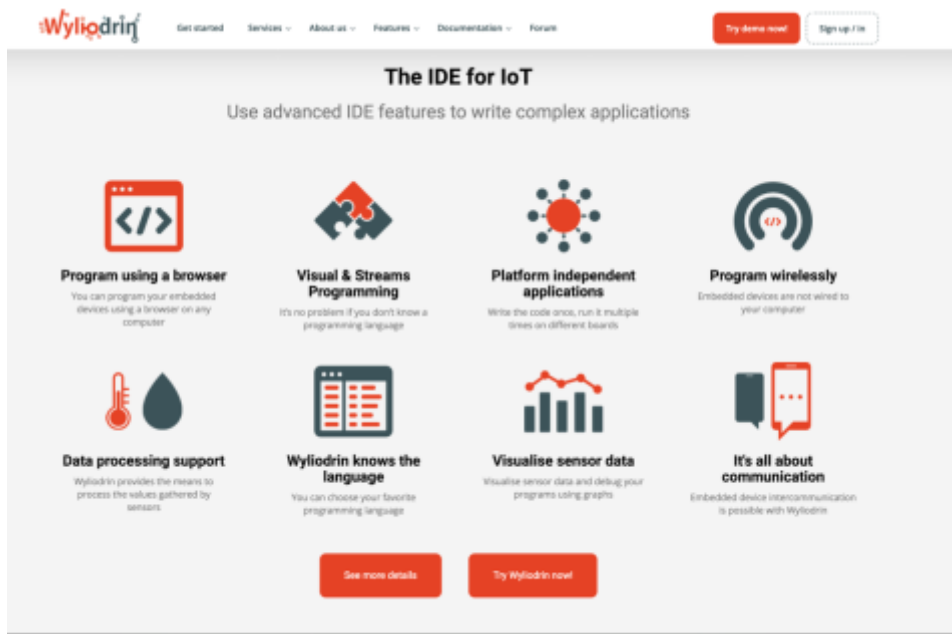
There is a gear menu at the top that we need to use for several important functions.



After you add your board, you are going to have to configure, get extra libraries, and download the wylodrin.json file. This JSON file must be put on your SD card.

What is wylodrin? It's an integrated development environment intended to serve the Internet of Things marketplace.

<https://www.wylodrin.com/>



Red Pitaya uses the wylodrin IDE to provide the Visual Programming platform.

Interestingly, Red Pitaya is not listed on the wylodrin site as one of the boards the downloaded version of the IDE supports. However, wylodrin does have a video tutorial that shows wylodrin visual programming in action.

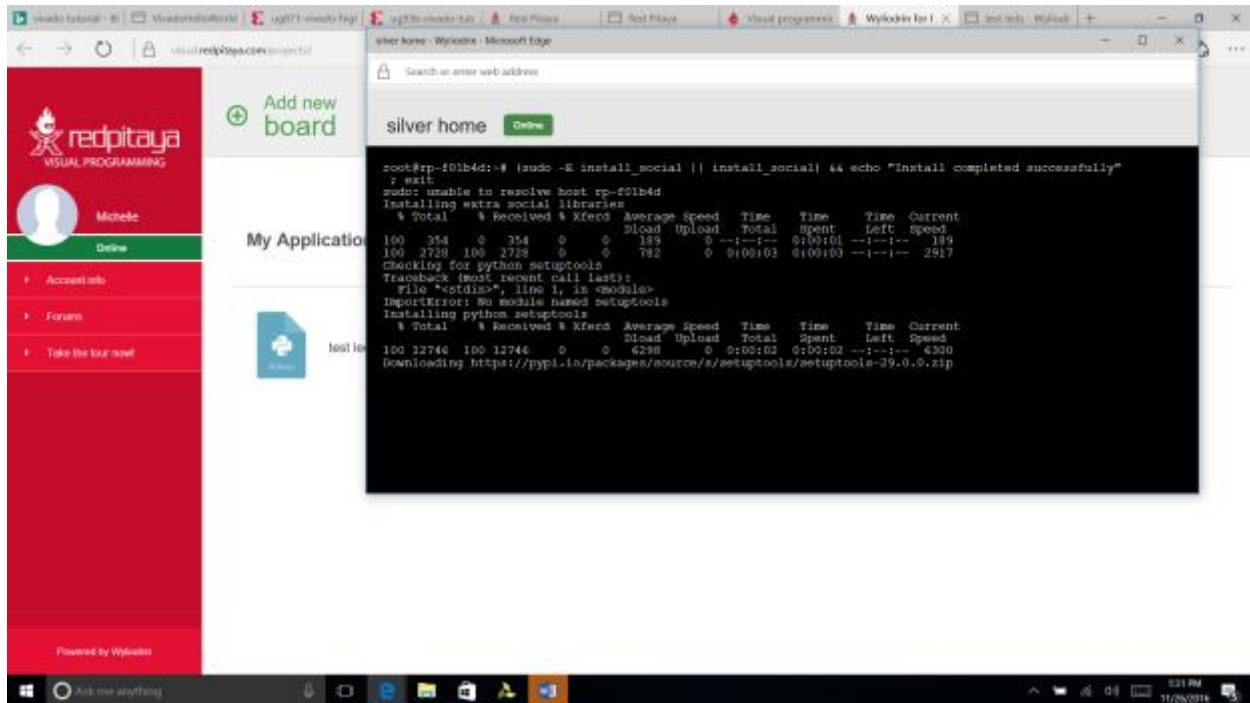
https://projects.wylodrin.com/wiki/video_tutorials/visual

The example video tutorial is a radio project on a Raspberry Pi. It's a radio with a VU meter. Not a bad "hello world" project. In wylodrin, the blocks (think of them as Legos) that form the visual programming are compiled into Python code. There's a pane for viewing this code within the IDE.

JSON stands for JavaScript Object Notation. JSON files are lists of attributes and value pairs. The lists can be gathered up into JSON objects and/or expressed as JSON arrays. Arrays and objects can be nested.

JSON files are fetched and key-value data extracted. JSON objects are used to create very powerful connected experiences. It's an open standard. Almost all of the applications for the Red Pitaya are enabled by JSON. Your web browser is the front-end, and your Red Pitaya is the back-end. Data between the Red Pitaya and the web browser is passed back and forth in JSON format. This is essentially the same strategy as Phase 4 Ground radios will use.

Here's what it looks like when you install extra libraries.



The Setup Tutorial link in the gear menu has all the steps we've discussed so far and a direct link to the JSON file.

Without this JSON file, you can click on the visual programming link all day long and nothing productive will happen, but it won't give you a useful error message. Ask me how I know.

In the process of handling the JSON file on a Windows machine, I found out about `brackets`, which is a "modern open source text editor that understands web design". I found it useful. It can be found here:

<http://brackets.io/>

The JSON file is human readable and as of mid-December 2016 was (for me) looked something like this.

```
{
  "jid": "abraxas3d silver home@visual.redpitaya.com",
  "password": "<REMOVED>",
  "socketpassword": "<ALSO_REMOVED>",
  "owner": "abraxas3d@visual.redpitaya.com",
  "timeout": 2000,
  "maxBuffer": 200,
  "firewall": false,
  "ping": 50,
  "ssid": "",
  "scan_ssid": 1,
  "psk": ""
}
```

This file controls relationships between your Red Pitaya board and several other entities.

The board will send status message to Red Pitaya servers for "improvement purposes". This can be turned off by adding a line to the wliodrin.json file.

Add

```
"privacy":true
```

Somewhere within the curly brackets.

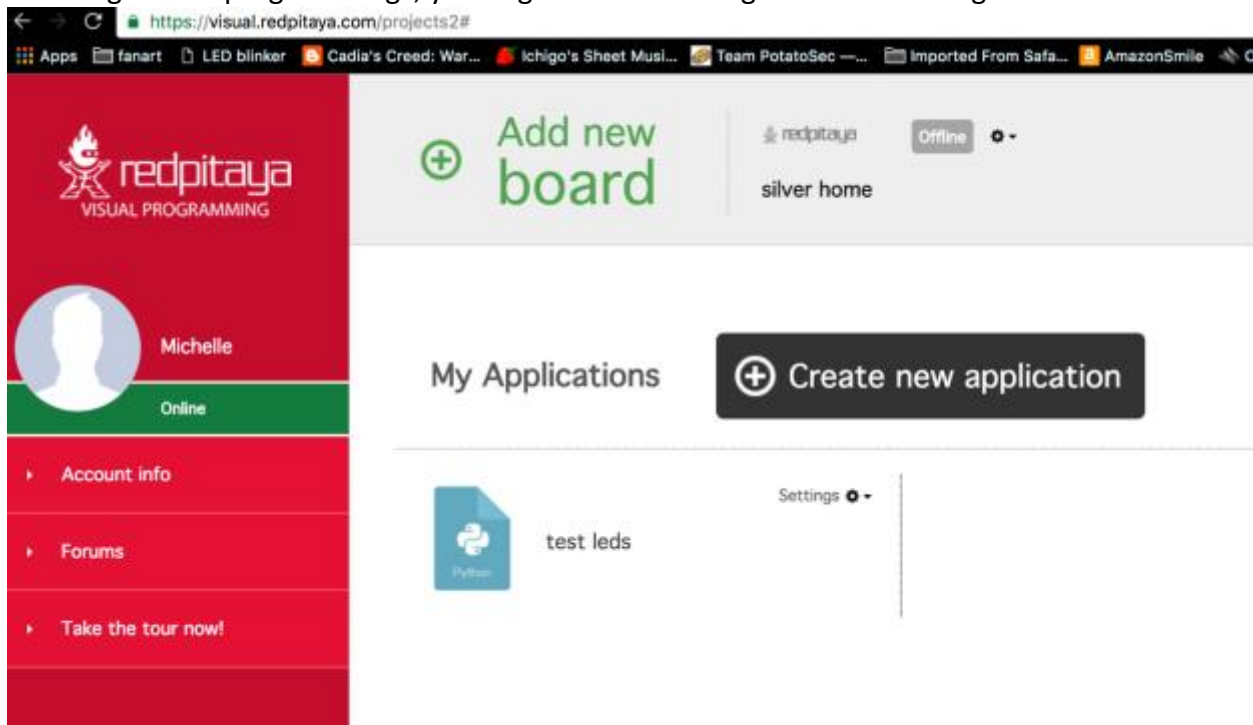
If only online privacy were always this easy everywhere.

What do the other lines of this JSON file do?

`jid` and `password` are XMPP connection credentials. Wylidrin servers use the listrophe library. The `jid` is used to send messages to the Red Pitaya. The board ID can be found in the gear menu option.

The other pairs of values seem to support common server-client communications functions. If I find out any additional functions that can be enabled through this file, I'll include them in a future revision. If you know of any, then please feel free to update this file or send me edits.

After the wylidrin.json file is on the SD card, and after everything is configured, and after selecting "visual programming", you might see something like the following screen



Notice in the upper middle-right part of the screen is a greyed-out box that says Offline, even though the green status indicator on the right-hand red-background menu says Online? We need to start the visual programming server. Go back to the My Equipment screen.

MY EQUIPMENT

Add New Board

Name	MAC address	LAN IP	Refresh time	WiFi IP	Refresh time	OS ver	OS build			
SilverHome	00:26:32:f0:1b:4d	192.168.1.149	Dec 14, 2016 2:31:25 AM	-	-	0.96	86	REMOVE	UNLOCK APPS	OPEN

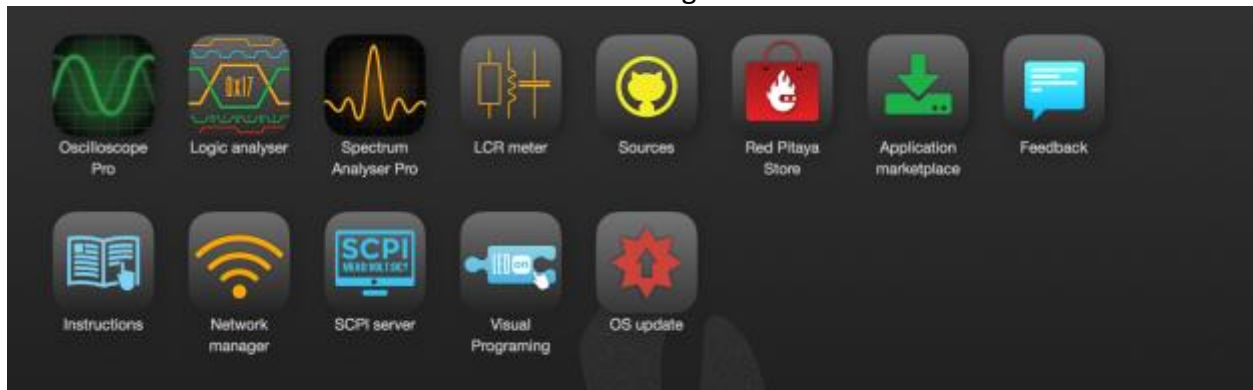
No application unlocked

Visual programming licence: Valid until 2017-04-03 22:14:17. Visit Visual programming [HERE](#).

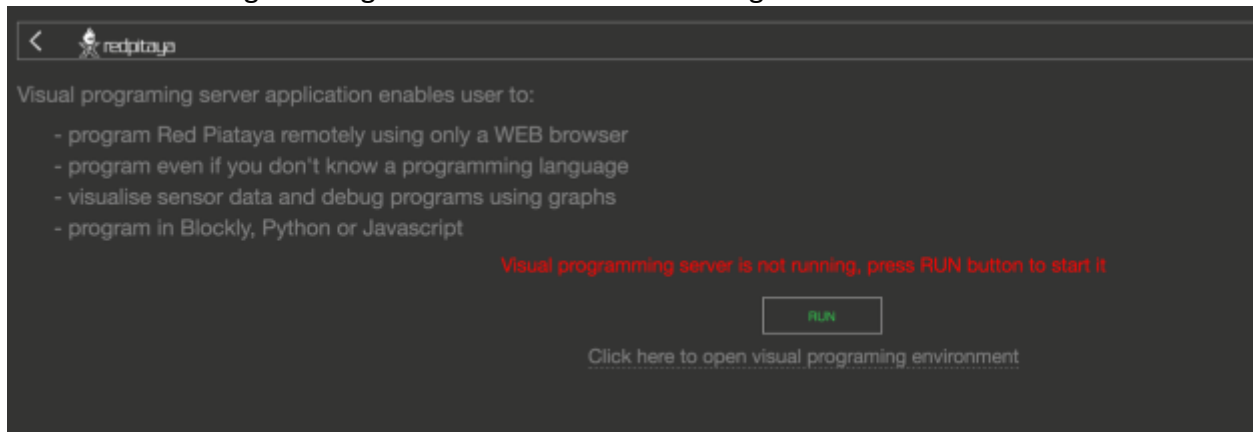
FAQ: Connectivity problems?

Where it says LAN IP, there is a link to the IP address of the Red Pitaya. If everything else has gone well, then the Red Pitaya is indeed sitting on your LAN, waiting to be told what amazing things it's going to be doing.

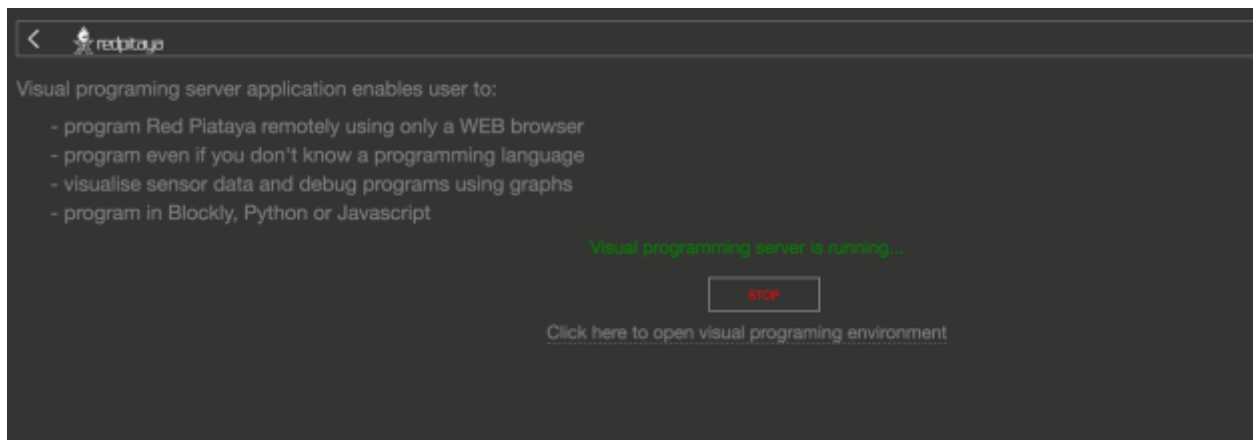
Click on that IP address. You should see the following screen.



Click on Visual Programming. You should see the following screen.

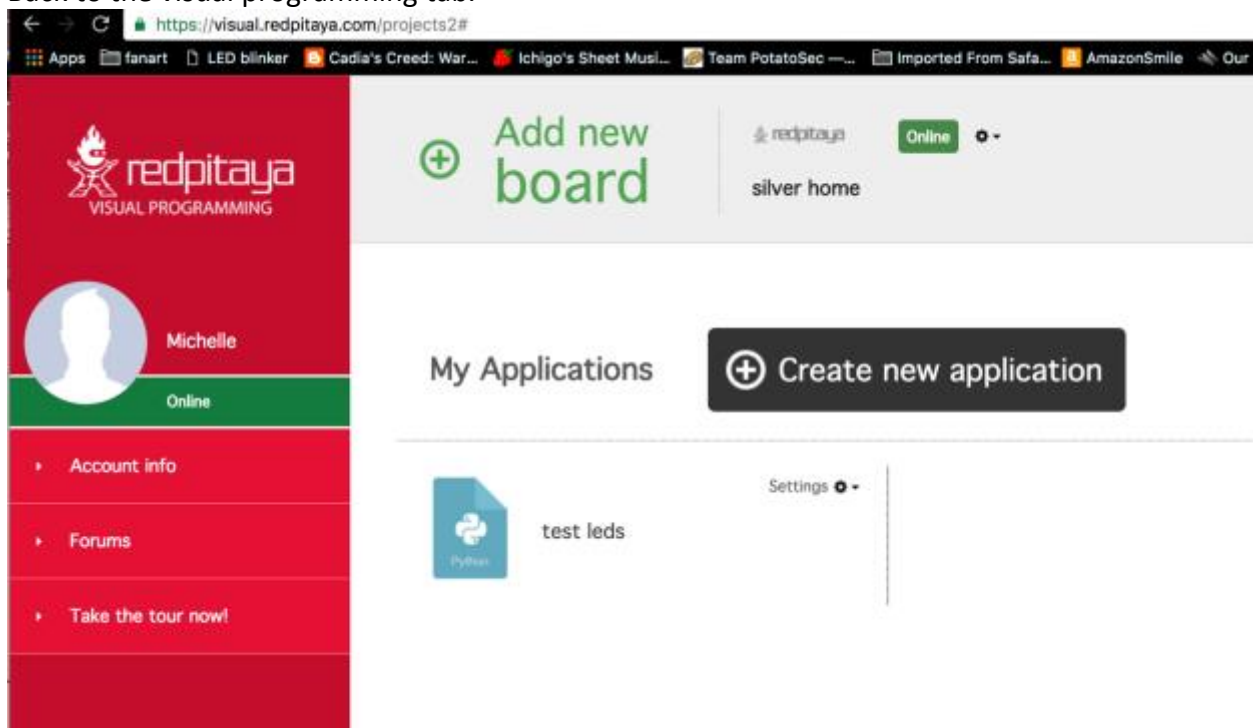


See where it says “Visual programming server is not running, press RUN button to start it”? Press the button.



OK that's promising!

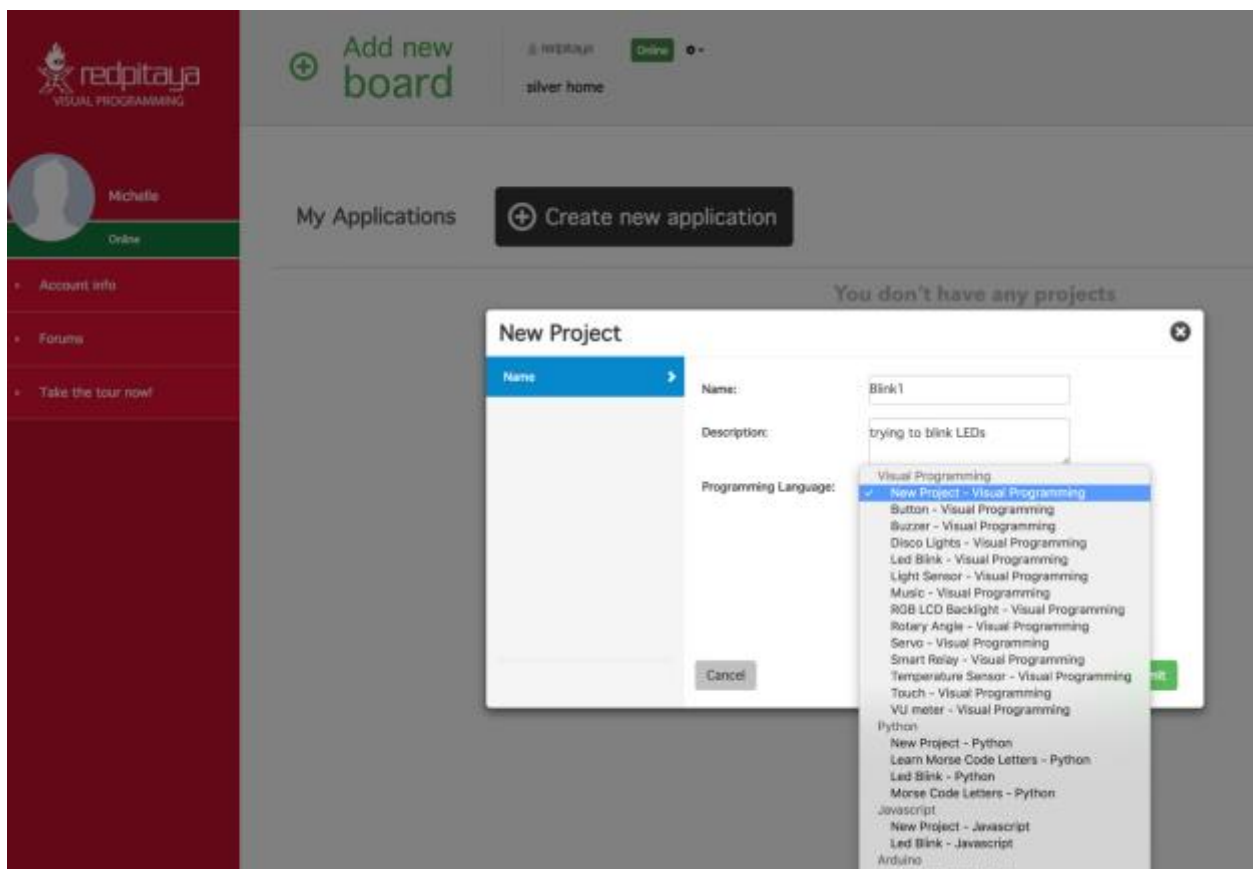
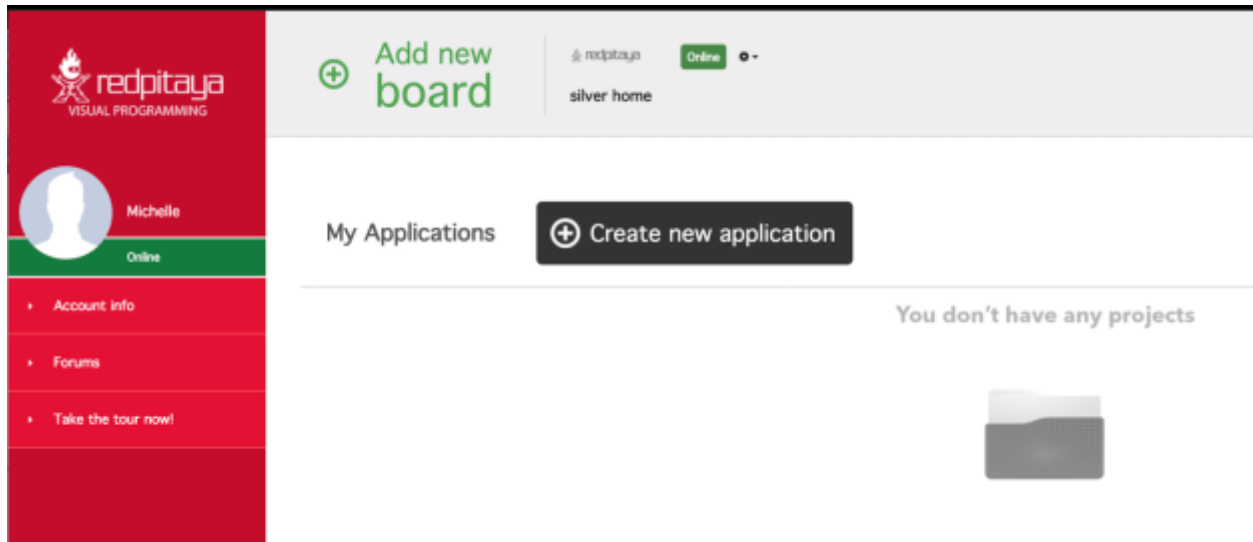
Back to the visual programming tab.



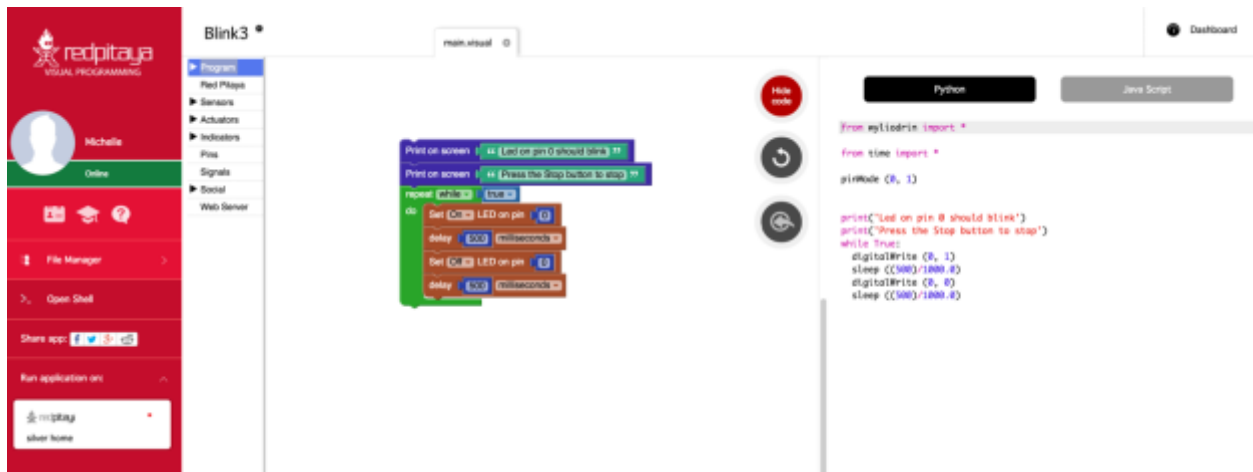
Improvement! The gray Offline box has turned to green Online. This is the point where we can get some traction from the published tutorials from wylodrin and Red Pitaya.

Programming With Visual Programming

Here's a simple LED blinking experiment within Visual Programming for the Red Pitaya. First, I created a new application. Select Create New Application and fill out the form.

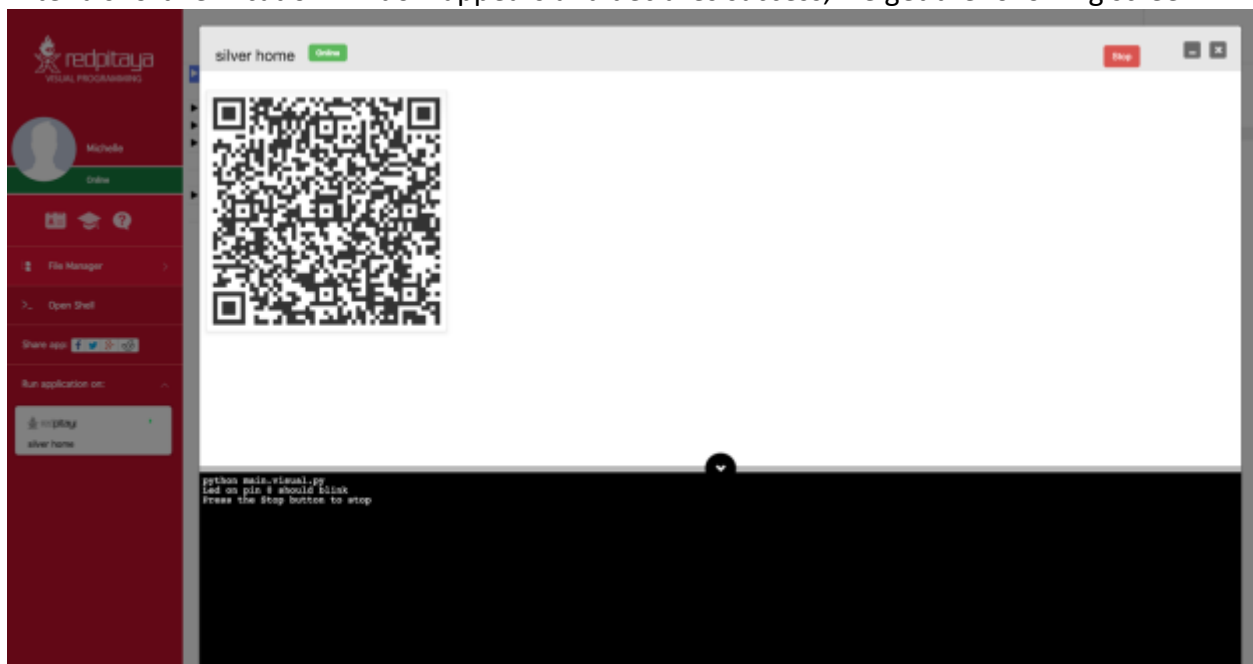


Well, there are a lot of choices in Programming Language. I went with New Project - Visual Programming. Look at all the template projects in the Visual Programming section! If you pick one, then you can see that they are the stock template projects from wylidrin. I created another project and opened up Led Blink. Here's what it looks like when you click Open Code button to show the Python pane.



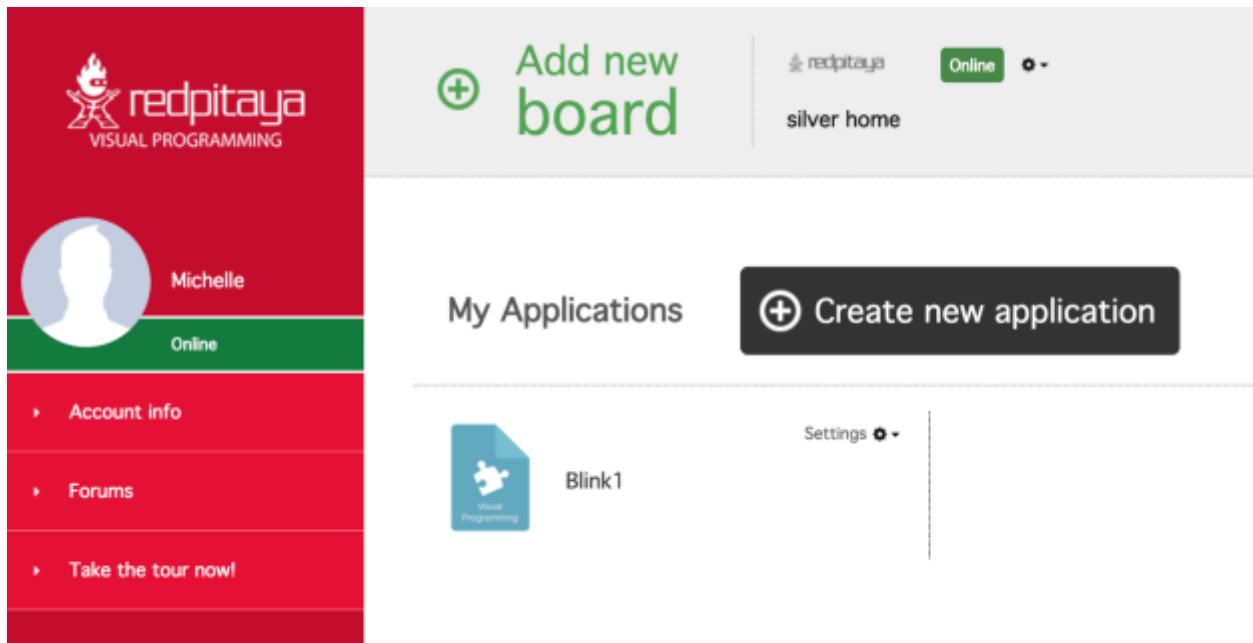
In the lower left corner is Run application on: and my Red Pitaya (silver home) shows up. I clicked this button.

After a short verification window appears and declares success, we get the following screen.

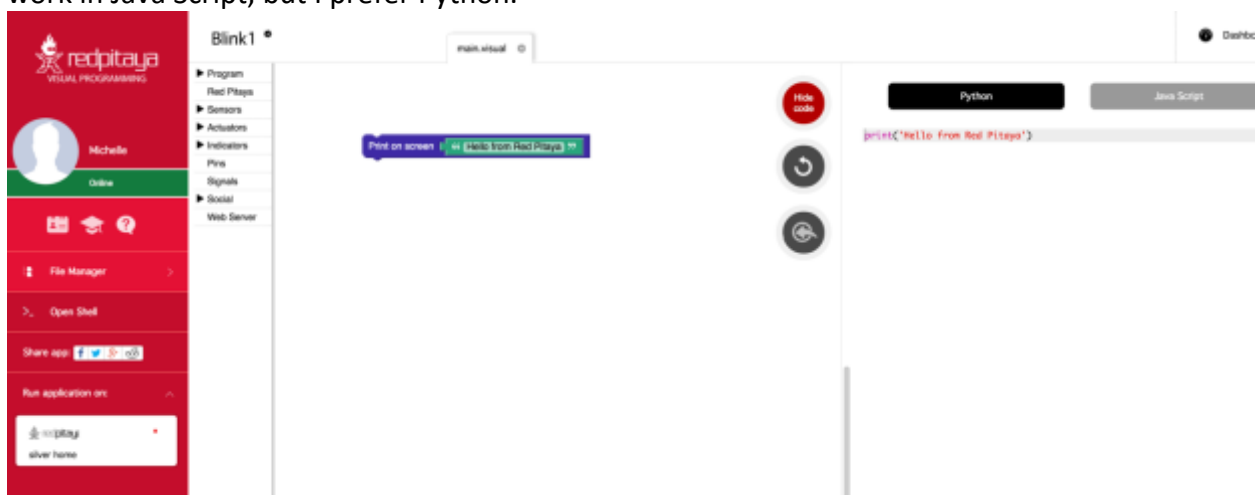


The 2D code is the board ID. You can see this same board ID in the gear menu. The stop button is in the upper right. The console declares that pin 0 should blink. I don't see anything blinking, but I haven't added an LED to the board or anything like that. Clicking stop made it stop claim that it had stopped. I was able to back out to the Visual Programming screen and the Red Pitaya was still online. Good progress!

When you Create new application, and select New Project – Visual Programming, you get a new project with a single line of code in it to start you off.



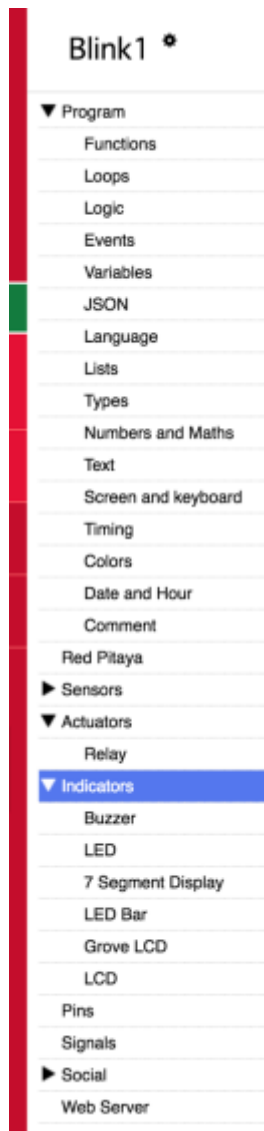
Here I created Blink1 and opened it, then clicked Show code to reveal the Python. You can also work in Java Script, but I prefer Python.



Running this code displays the output of `print('Hello from Red Pitaya')` in the console, as expected.

Now, where do you get those Lego-like blocks that you use to build programs? Immediately to the right of the red menu on the left-hand side is a column of menus, with Program, Red Pitaya, Sensors, Actuators, Pins, Signals, Social, and Web Server.

Those drop downs are where the programming blocks are located.



So, for our first program, we're going to need a few blocks. In thinking about the instructions to flash LEDs, we know we're doing a simple task over and over. An obvious way to accomplish this is with a Repeat block. This will cause continuous executions of everything which is inside the block. Think of it as if it was a while loop. Inside the Repeat block we are going to have two Set LED blocks. These blocks switch ON and OFF the LEDs. Between switching the LEDs on and off, we need time delays so that we can see the LEDs with our boring old human eyes.

What are the advantages of using Visual Programming?

According to Red Pitaya HQ, advantages include:

Ability to create own dashboards with real time graphs, dials, meters, sliders, and buttons.

Ability to control the program flow from a PC, smartphone or tablet.

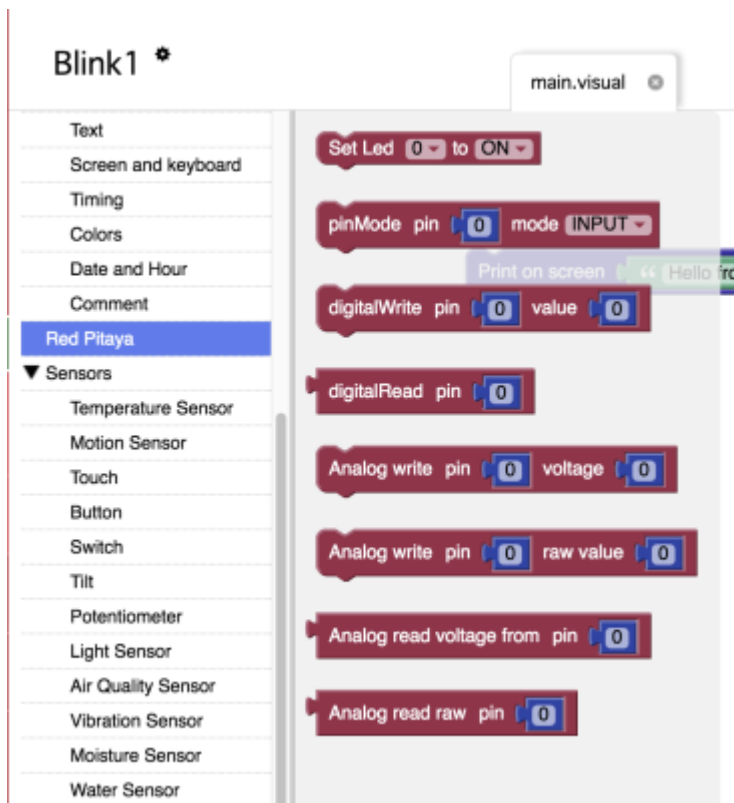
Ability to share measurements or send notifications to email or even social networks like Facebook and Twitter.

Measures temperature, moisture, alcohol, water level, vibrations, UV light, sound, pressure, air quality detect motion, and others.

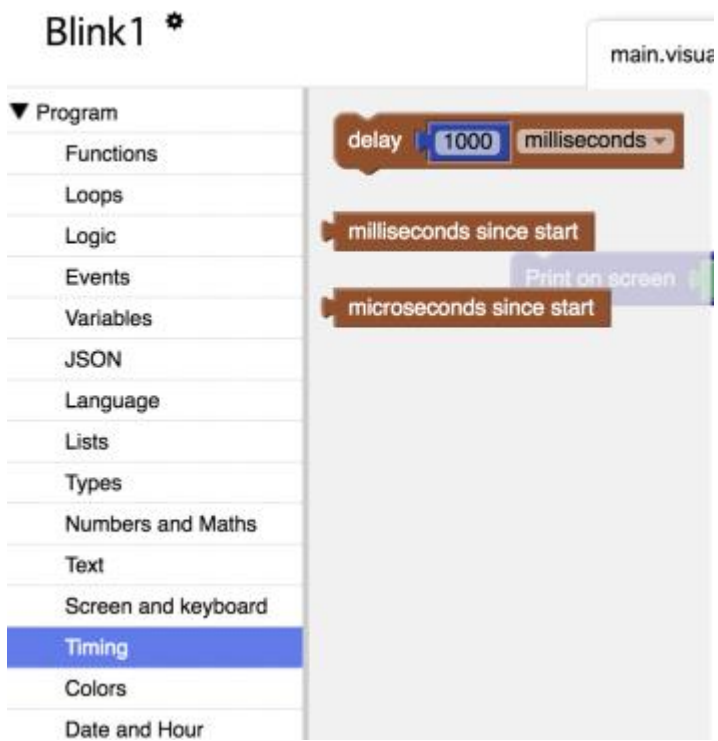
Control actuators and indicators like LEDs, displays, motors or relays in order to control high load devices (requires the use of the Red Pitaya Sensor extension module & sensors)

Programming with blocks is a very fun experience, but is also highly instructive and encourages the user to begin thinking subconsciously like a real programmer.

This format also enables users to watch and learn what the real programming language code behind the graphical blocks looks like – and how to program using it.



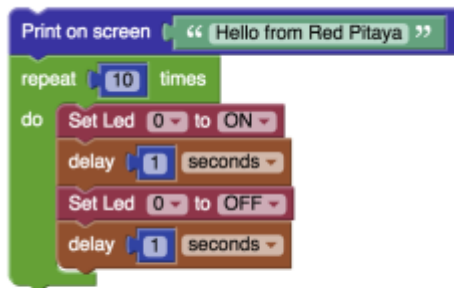
The set LED function is in the Red Pitaya menu (of course).



The “delay some amount of unit time” block is the Program menu under Timing.



The loop control blocks are in Program under Loops. For this project I picked repeat X times block. At the time of this writing, it was the second block from the top in this section. Here's what it looks like when you click all the blocks together and configure each of them to make blinking happen.

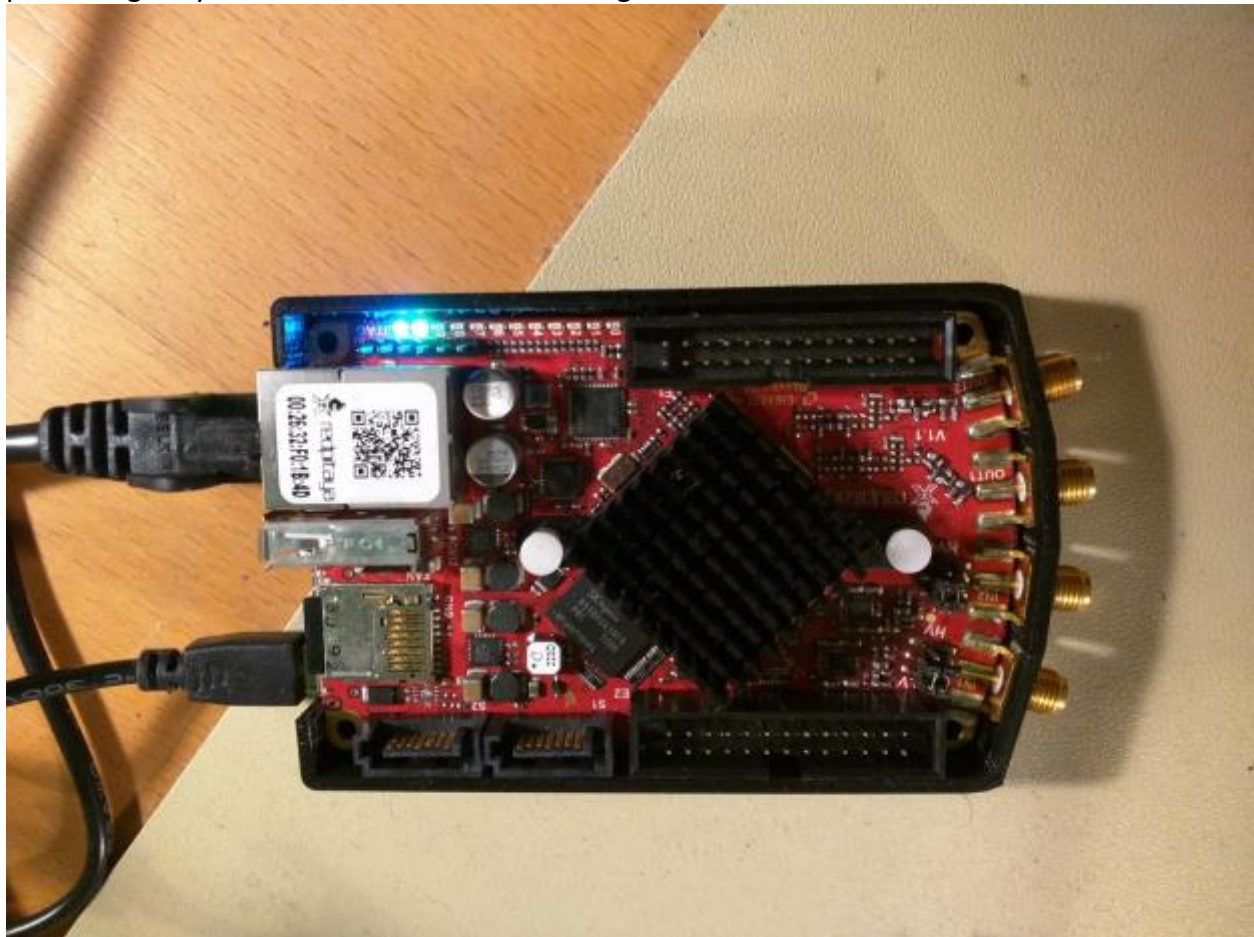


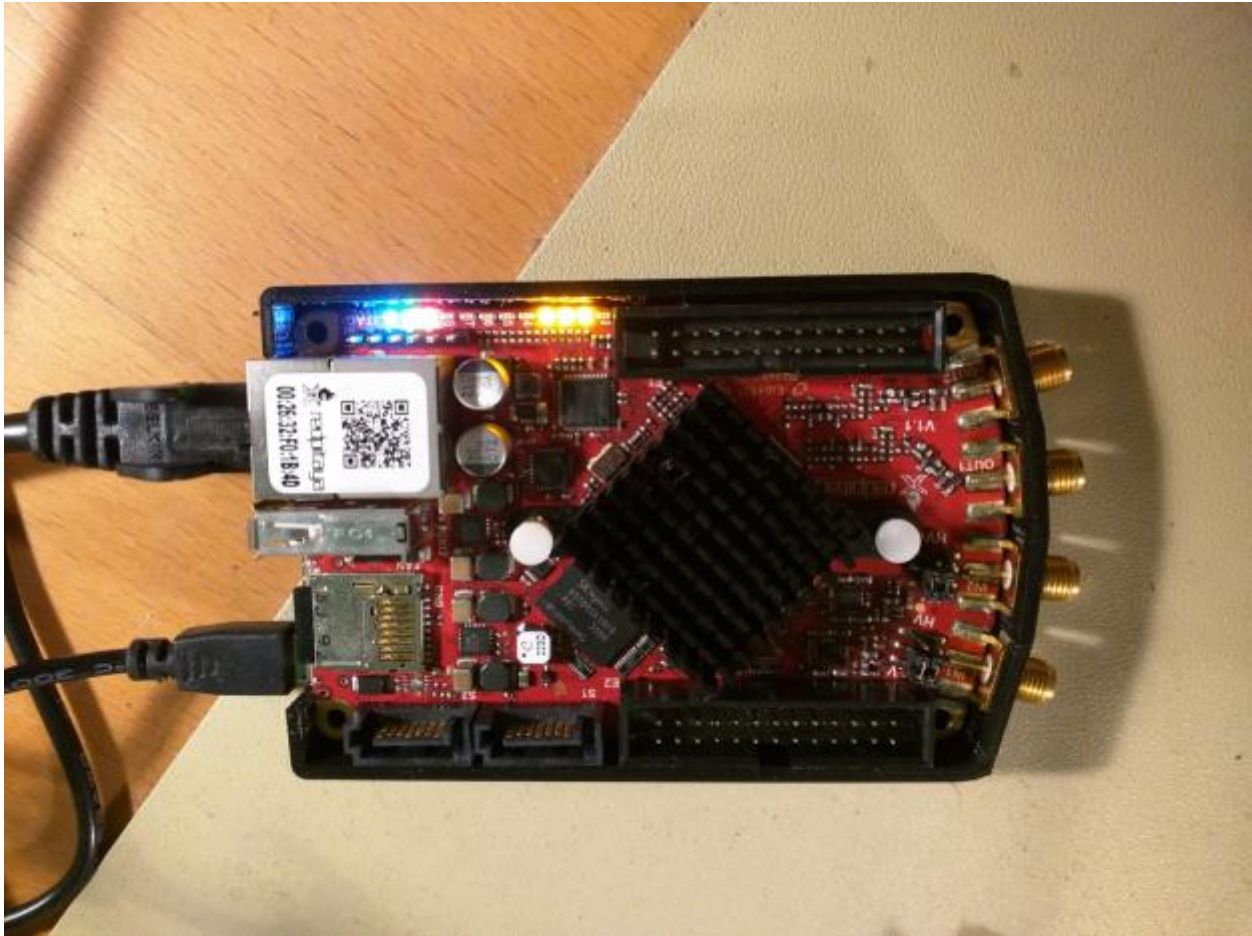
Not being satisfied with one, I decided to turn on and off three! Also, I changed it to run forever.



The drop-down menu in the Set LED block for LED number has the choices 0 through 7. I selected 3, 2, and 1 for the second blinking LED exercise. What do these numbers correspond to?

If you look on your Red Pitaya board, you can see 8 LEDs along the edge next to the Ethernet connector. Each LED corresponds to one of the 0 through 7 LEDs references in the Set LED block. They are built into the Red Pitaya board. The LEDs on the board have their number printed right by them on the silk screen. Can't get much easier than that.





Here's the three LEDs on.

There's more to do with Visual Programming that just turn LEDs on. Grove support provides the horsepower behind the Sensors and Actuators menus, for example.

But, Visual Programming is limited. The Visual Programming interface provides pre-existing blocks. These blocks give high-level access to a fixed set of registers. There isn't a way to reprogram the FPGA from Visual Programming. Visual Programming does not have blocks that let you access the Fast Analog outputs and inputs.

Despite the limitations, Visual Programming can produce a wide variety of Internet of Things functionality. Interactive functions under the Social menu include blocks such as mail, Facebook, Twitter, Twilio, and Board Communication. You can set up Web Server functions as well. Leveraging the power of the wylodrin IDE was a good choice and provides useful functionality. What it doesn't seem to provide is access to the FPGA.

We'll have to get a lot closer to the hardware in order to fully achieve the goals of Step 2!