



Introduction

This is a guide on how to set up an FPGA ([Intel DE10-Nano](#)) to run the long awaited [IOTA Qubic](#) Proof-of-Concept. It will teach you how to generate a bitstream ([.rbf](#)) from the [HDL](#) source code, load this bitstream into the FPGA, load and run an [Abra](#) config file, and extract the results via HTTP.

What is the QLUT?

The QLUT is a data-flow processor composed of LUTs (look-up tables) that can be configured to run Qupla functions remotely on dedicated hardware. The hardware communicates with a host computer via a TCP/IP socket interface where configuration data, inputs, and outputs are sent back and forth to the client and server.

Required Software:

Please use the exact versions detailed here if applicable. Intel's tools are very sensitive to version changes and it is very likely things will not work if you do not use the right version

- Quartus Prime Lite 18.1
- Host PC running Windows 10 or Linux OS (Ubuntu 16.04 recommended)
- QLUT Source Files (available at github.com/iotaedger/qubic-hdl)
- [PuTTY](#) or similar SSH program
- Software to write disk images ([Win32DiskImager](#) recommended for Windows Host)
- DE10-Nano GHRD (available from the board's revision C [system CD](#))
- [Qupla source](#) (follow README on page)

Required Hardware

- Cyclone V DE10-Nano SoC
- Mini USB to USB cable (for UART connection)
- 5V (2A) DC power supply
- 2GB+ microSD card
 - Adapter to interface microSD with Host PC

Setting Up Intel Quartus Prime Lite 18.1

For the purposes of this guide, we won't be directly covering how to install all the required software to run the PoC since Intel already has guides on how to install their software, and there's a plethora of resources on the internet on how to install an OS.

Please refer to Chapter 3.2 in Intel's [DE10-Nano Getting Started Guide](#) for installation instructions. All default paths and installation settings should suffice; make sure to include Quartus Lite, Modelsim Free Edition, and Cyclone V device support. Also install any drivers if prompted with their installation.

Another really good resource is Intel's [terasic-de10-nano-kit](#) repository. It has 6 tutorials to help you familiarize yourself with the DE10-Nano kit.

Downloading the QLUT Source Files

The following github repository contains the QLUT source files:

<https://github.com/iotaedger/qubic-hdl>

The following git command can also be used to acquire the source files:

`git clone https://github.com/iotaedger/qubic-hdl.git`

The QPS (QLUT Processor System) [source](#) has the following file structure:



- /Firmware
 - /ip
 - /PLL_0
 - PLL_0_0002.qip
 - PLL_0_0002.v
 - /QLUT_AVALON
 - QLUT_2B_3IN.vhd
 - QLUT_AVALON.vhd
 - QUBIC_PROCESSOR.vhd
 - DE10-NANO_SoC_GHRD.v
 - PLL_0.qip
 - PLL_0.sip
 - PLL_0.v
 - QLUT_AVALON_hw.tcl
 - soc_system.qsys
 - soc_system.rbf
- /FpgaServer
 - /server
 - FpgaServer.c
 - Layout.c
 - Layout.h
 - Qlut.c
 - Qlut.h

Downloading the DE10-Nano GHRD

******If you would like to use the system without generating the bitstream from the source, modifying the source, or studying its underlying mechanisms, and are only interested in seeing the hardware provide you a result for a Qupla function, you can skip down to “Setting up a Linux OS on the DE10-Nano”******

The golden hardware reference design (GHRD) is what Altera calls its pre-configured hardware design that is ready for engineers to play with. It comes pre-built with utilities that allow you to interact with peripherals on the board when running a linux OS on it. We will not be going into the GHRD in detail, but we will need it as a baseline project for this tutorial. To download the GHRD, navigate to this [link](#) and choose the revision C system CD under the “CD-ROM” table.

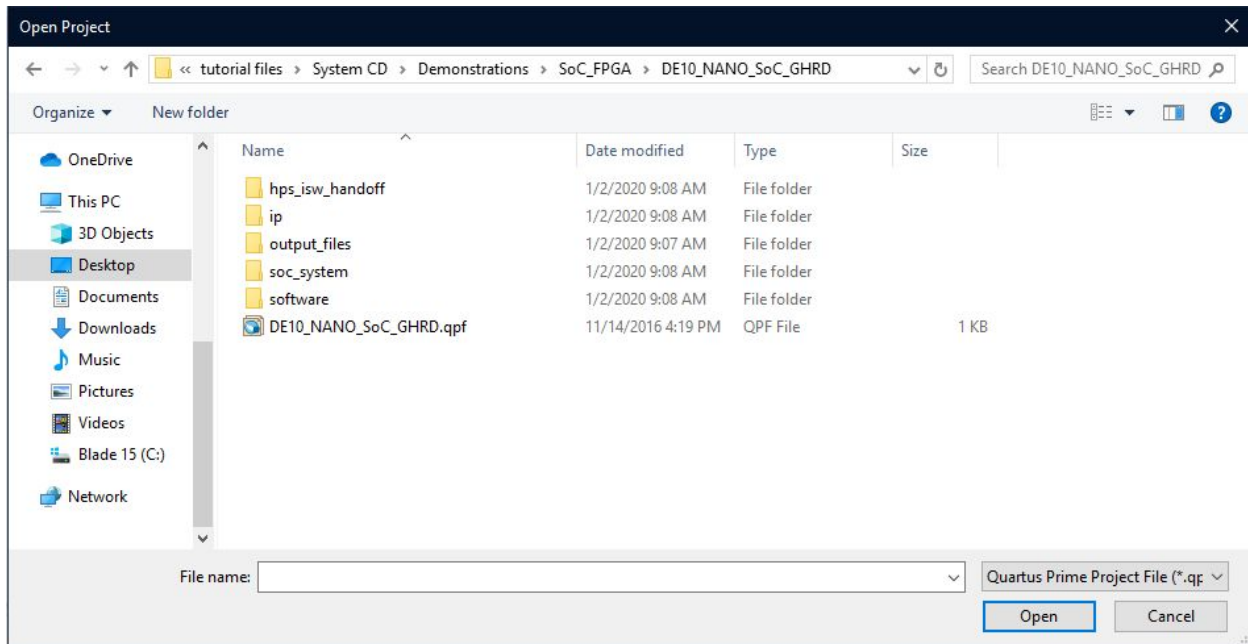
CD-ROM

Title	Version	Size(KB)	Date Added	Download
DE10-Nano CD-ROM (rev. C Hardware)	1.3.8		2020-01-02	
DE10-Nano CD-ROM (rev. A/B Hardware)	1.2.4		2018-02-01	
DE10-Nano CD-ROM (rev. B2 Hardware)	1.2.5		2018-02-01	
Network Socket Example Design	1.0.0		2017-10-30	
Quartus Download	16.0		2016-12-22	

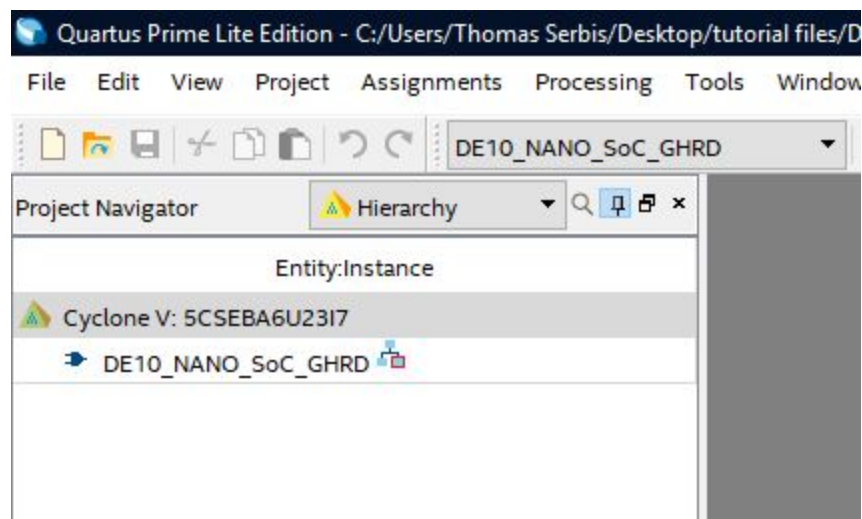
You will need to register an account with Terasic to download the file, which won't be covered in this tutorial, but it's not very difficult. Once you download the zip folder, create a folder to extract it to and extract it to that folder.

Setting Up the Quartus Project

Quartus Prime is Intel's development environment for creating hardware designs using hardware descriptive languages, such as Verilog or VHDL, and as such, it requires setting up an environment for each project. To do so, open up Quartus Prime Lite, and navigate to the top left. Click on *File < Open Project*. We now need to navigate to the folder containing the GHRD.



The path should be `/Rootfolder/Demonstrations/SoC_FPGA/DE10_NANO_SoC_GHRD`. The file we want to open here is the Quartus project file, `DE10-NANO_SoC_GHRD.qpf`. When you successfully open the file, you should see the following in the top left of the main Quartus window (NOTE: if you see an option to launch the IP upgrade tool at any time during this tutorial, upgrade the IP):



You have successfully opened the GHRD, but we still need to modify it by adding the QLUT processor system (QPS). Next we will dive into copying the QPS source into the Quartus project directory.

Adding QPS Source to the Quartus Project Directory

We will need to move the following folder and files into our Quartus project folder from the QPS source directory (*make sure to replace all files when asked*):

- /PLL_0
- DE10-NANO_SoC_GHRD.v
- PLL_0.qip
- PLL_0.sip
- PLL_0.v
- QLUT_AVALON_HW.tcl
- soc_system.qsys

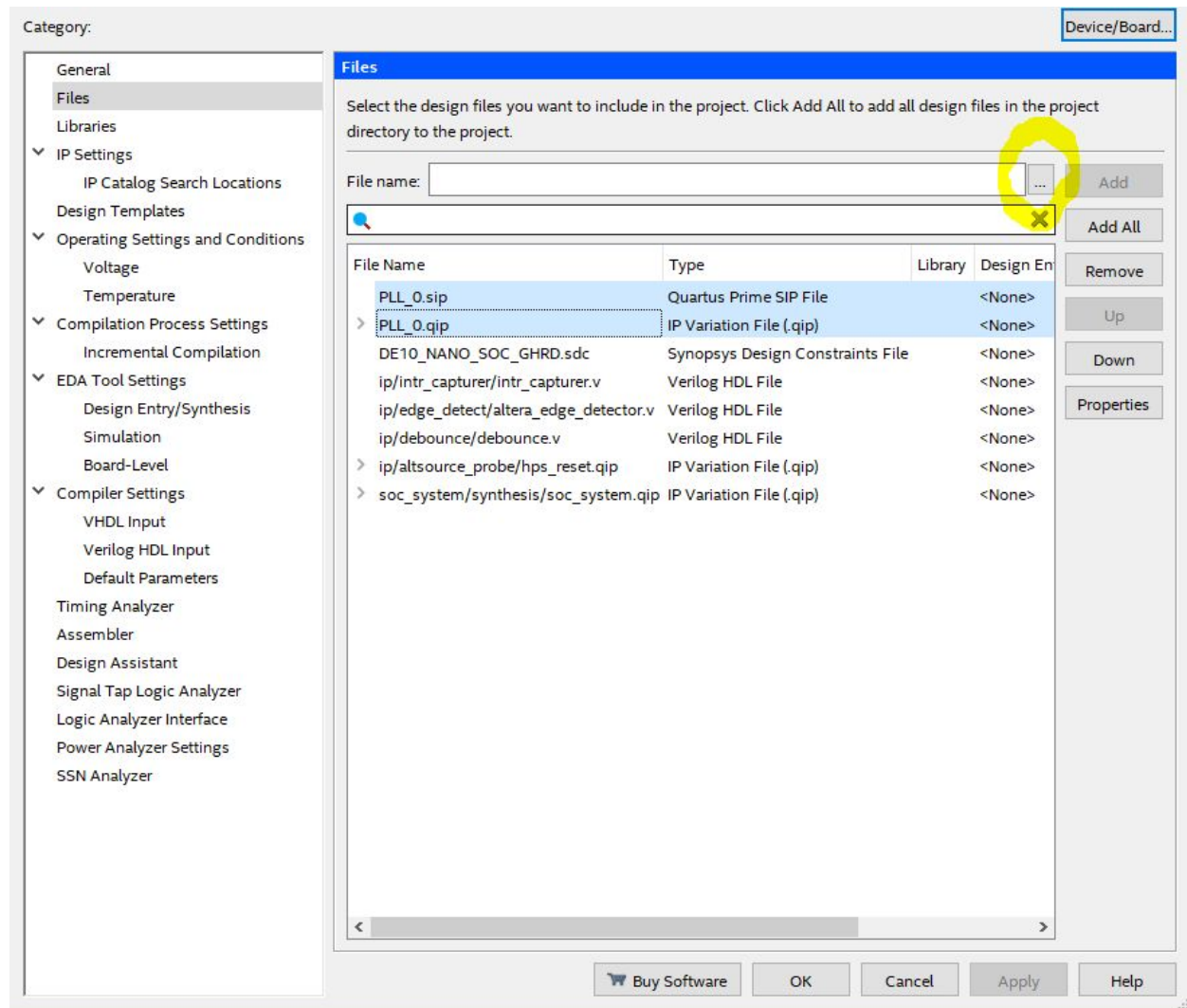
incremental_db	3/27/2020 12:12 PM	File folder	
ip	3/27/2020 9:39 AM	File folder	
output_files	3/27/2020 1:15 PM	File folder	
PLL_0	3/27/2020 12:11 PM	File folder	
soc_system	3/27/2020 10:03 AM	File folder	
software	3/27/2020 8:02 AM	File folder	
c5_pin_model_dump.txt	3/27/2020 12:34 PM	Text Document	5 KB
DE10_NANO_SoC_GHRD.ipregen.rpt	3/27/2020 7:59 AM	RPT File	59 KB
DE10_NANO_SoC_GHRD.qpf	11/14/2016 4:19 PM	QPF File	1 KB
DE10_NANO_SoC_GHRD.qsf	3/27/2020 12:11 PM	QSF File	43 KB
DE10_NANO_SOC_GHRD.sdc	11/25/2016 12:21 ...	SDC File	3 KB
DE10_NANO_SoC_GHRD.v	3/27/2020 8:50 AM	V File	16 KB
DE10_NANO_SoC_GHRD_assignment_def...	3/27/2020 7:42 AM	QDF File	55 KB
generate_hps_qsys_header.sh	12/5/2016 11:22 AM	SH Source File	1 KB
hps_common_board_info.xml	4/10/2017 11:47 AM	XML Source File	14 KB
hps_sdram_p0_all_pins.txt	3/23/2017 10:52 AM	Text Document	7 KB
hps_sdram_p0_summary.csv	3/27/2020 12:37 PM	Microsoft Excel C...	2 KB
Makefile	11/25/2016 12:21 ...	File	20 KB
PLL_0.qip	3/27/2020 11:49 AM	QIP File	52 KB
PLL_0.sip	3/27/2020 11:49 AM	SIP File	1 KB
PLL_0.v	3/27/2020 11:58 AM	V File	18 KB
QLUT_AVALON_hw.tcl	3/27/2020 8:49 AM	TCL File	12 KB
soc_system.BAK.qsys	3/27/2020 7:54 AM	QSYS File	50 KB
soc_system.BAK.v	3/27/2020 7:54 AM	V File	106 KB
soc_system.dtb	4/10/2017 11:47 AM	DTB File	24 KB
soc_system.dts	4/10/2017 11:47 AM	DTS File	46 KB
soc_system.qsys	3/27/2020 8:50 AM	QSYS File	48 KB
soc_system.rbf	3/27/2020 12:58 PM	RBF File	6,843 KB
soc_system.sopcinfo	3/27/2020 10:02 AM	SOPCINFO File	2,655 KB
soc_system_board_info.xml	11/25/2016 12:21 ...	XML Source File	1 KB

Lastly, we must merge the */ip* folder in our source with the one in our Quartus project folder.

altsource_probe	3/27/2020 8:02 AM	File folder
debounce	3/27/2020 8:02 AM	File folder
edge_detect	3/27/2020 8:02 AM	File folder
QLUT_AVALON	3/27/2020 9:39 AM	File folder

Once this step is complete, you will have successfully moved the QPS source to your project directory.

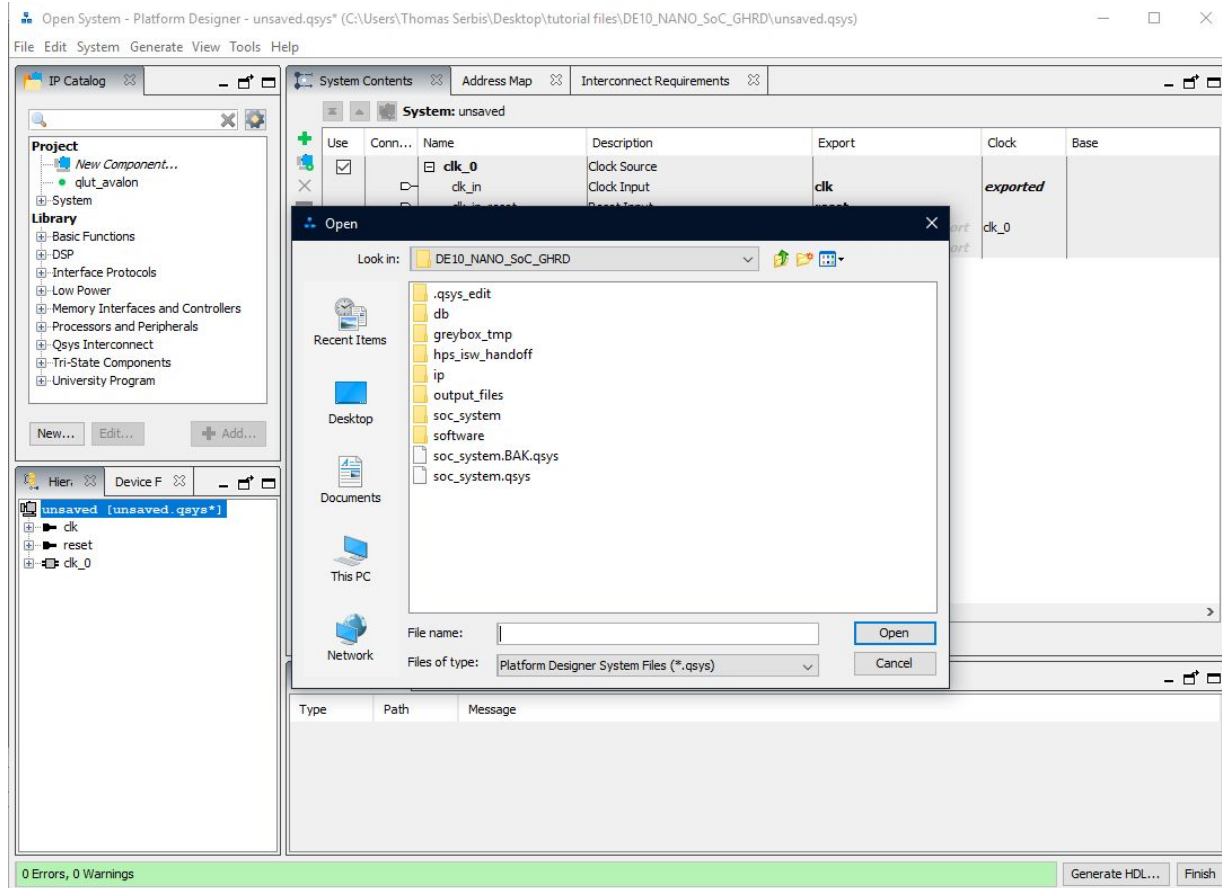
We will finish this section by adding *PLL_0.sip* and *PLL_0.qip* to our project. To do so, you will need to navigate to *Project < Add/Remove Files in Project* in the main Quartus window. You will want to click “...” in the top-right.



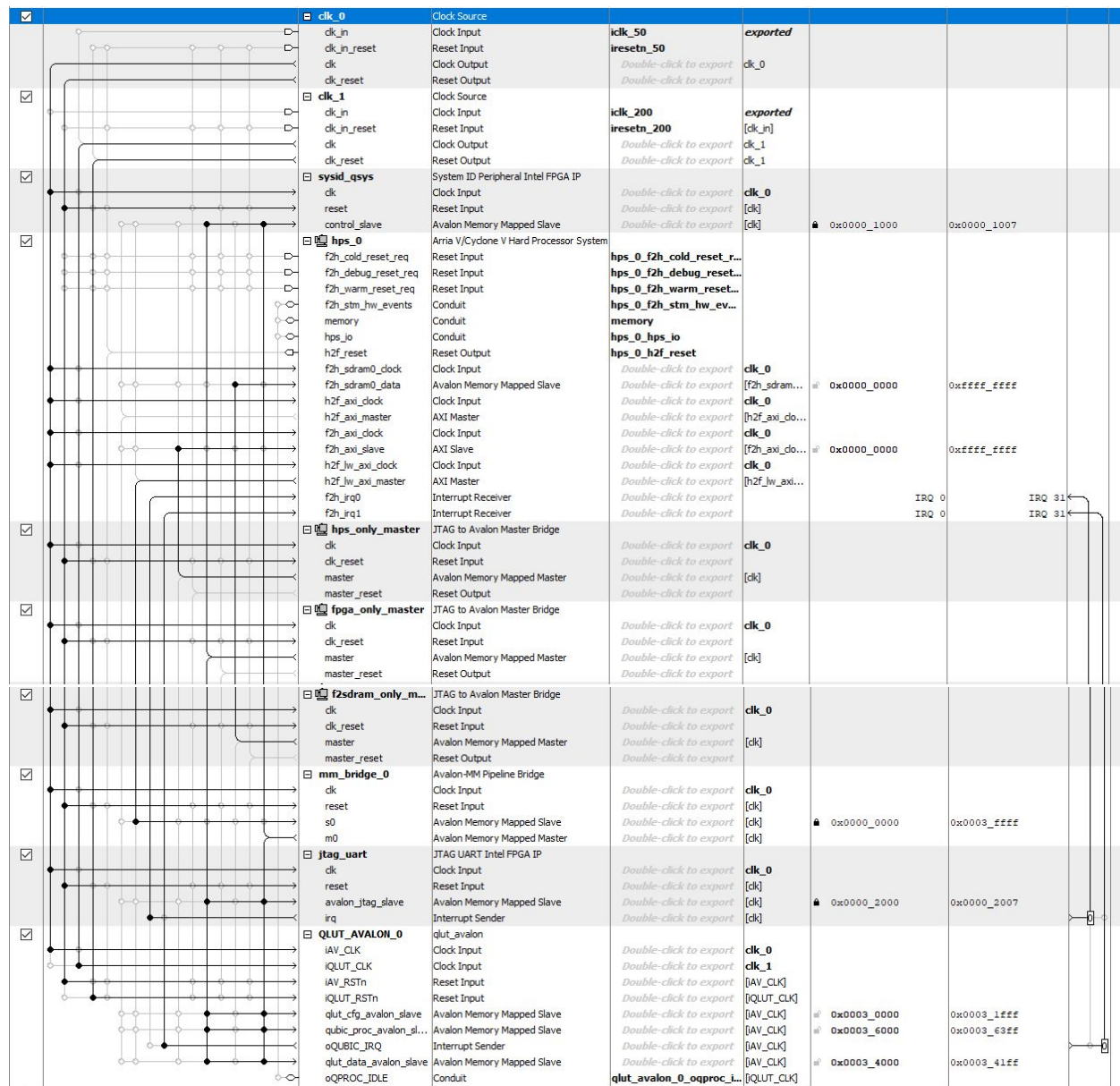
Find *PLL_0.sip* and *PLL_0.qip*, add them, and then click *Apply* and then *Ok*. We can now move onto setting up the hardware system in Platform Designer.

Setting up the Hardware System in Platform Designer

To start, navigate to *Tools < Platform Designer* in the main Quartus window. The following window should pop up:



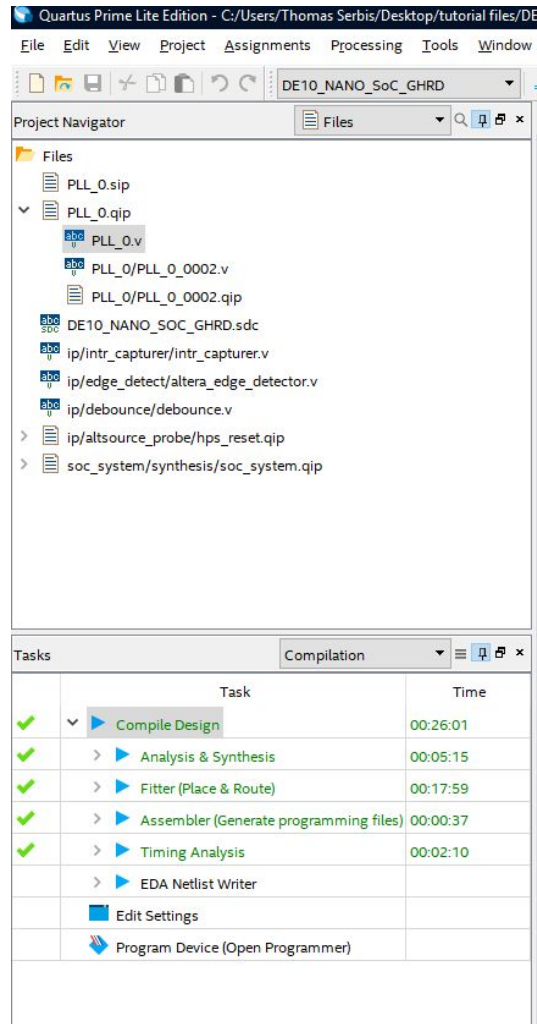
We want to open *soc_system.qsys*. This file describes the connections of our FPGA design with the HPS (hard-processor system) on the board. You should see the following after opening the file:



All we have to do now is generate HDL code from this design. To do so, click *Generate < Generate HDL* at the top of the Platform Designer window. In the pop-up window, keep the default settings and hit *Generate*. This process should take a few minutes. Once it's complete you should see that the generation completed with warnings (which is okay). We are now ready to compile the hardware system.

Compiling the Hardware System

When you get to this point, you should have the file structure in the following screenshot in the top-left of your Quartus window. If you do not, make sure to carefully follow all previous steps.



Compiling is as easy as double-clicking “Compile Design” on the bottom left of the window. The whole process will take anywhere from 25-35 minutes depending on the speed of your CPU. You will see a plethora of warnings, but they can be ignored. Once the compilation is finished we can move on to generating a .rbf for our design.

Generating a Raw Binary File Describing the Hardware System

To generate a raw binary file of the QPS design, navigate to */quartusproject/output_files* and double-click *sof_to_rbf.bat*. You should see a command prompt window like the following:

```
C:\Windows\system32\cmd.exe
C:\Users\Thomas Serbis\Desktop\tutorial files\DE10_NANO SoC_GHRD\output_files>C:\intelFPGA\18.1\quartus\bin64\quartus_cpf -c -o bitstream_compression=on DE10_NANO SoC_GHRD.sof soc_system.rbf
Info: *****
Info: Running Quartus Prime Convert_programming_file
Info: Version 18.1.0 Build 625 09/12/2018 SJ Lite Edition
Info: Copyright (C) 2018 Intel Corporation. All rights reserved.
Info: Your use of Intel Corporation's design tools, logic functions
Info: and other software and tools, and its AMPP partner logic
Info: functions, and any output files from any of the foregoing
Info: (including device programming or simulation files), and any
Info: associated documentation or information are expressly subject
Info: to the terms and conditions of the Intel Program License
Info: Subscription Agreement, the Intel Quartus Prime License Agreement,
Info: the Intel FPGA IP License Agreement, or other applicable license
Info: agreement, including, without limitation, that your use is for
Info: the sole purpose of programming logic devices manufactured by
Info: Intel and sold by Intel or its authorized distributors. Please
Info: refer to the applicable agreement for further details.
Info: Processing started: Fri Mar 27 13:15:12 2020
Info: Command: quartus_cpf -c -o bitstream_compression=on DE10_NANO SoC_GHRD.sof soc_system.rbf
Info (292036): Thank you for using the Quartus Prime software 30-day evaluation. You have 6 days remaining (until Apr 02
, 2020) to use the Quartus Prime software with compilation and simulation support.
Info: Quartus Prime Convert_programming_file was successful. 0 errors, 0 warnings
Info: Peak virtual memory: 4441 megabytes
Info: Processing ended: Fri Mar 27 13:15:14 2020
Info: Elapsed time: 00:00:02
Info: Total CPU time (on all processors): 00:00:02

C:\Users\Thomas Serbis\Desktop\tutorial files\DE10_NANO SoC_GHRD\output_files>pause
Press any key to continue . . .
```

You should see a file named `soc_system.rbf` in the same directory as the batch script. We will be using this file to load the QPS design onto the FPGA as a bitstream. Our next step is to set up a microSD card with our hardware design and a Linux OS.

Setting up a Linux OS on the DE10-Nano

Setting up a Linux OS is very straightforward if you follow [bernardoaraujo's guide](#) on [gist.github.com](#). The steps for flashing the microSD card differ slightly if you are developing on Windows vs. Linux.

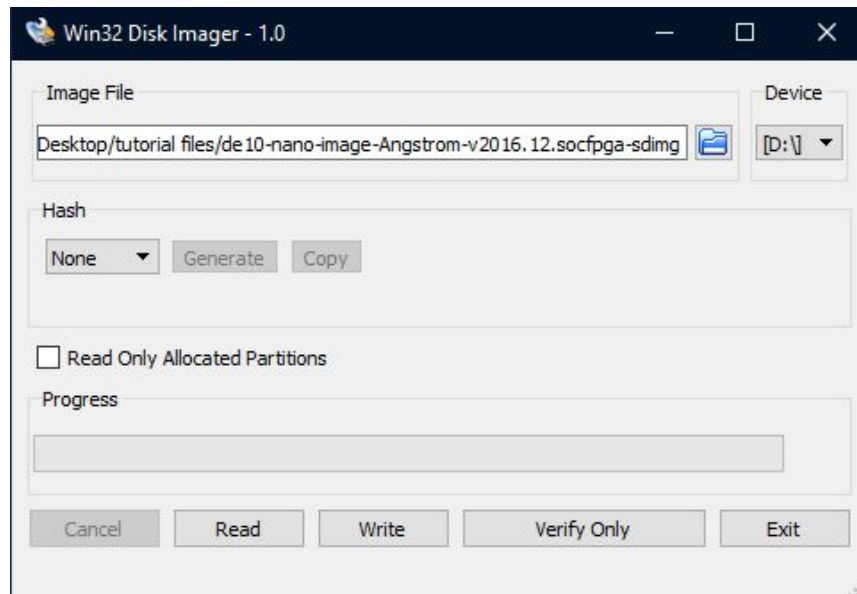
Linux Host PC Users

Linux users can follow the gist guide verbatim up until “Compiling devmem utility to interact with LUTs”. The steps from then on are not useful for the tutorial. Linux users can skip the next section, “Preparing the SD Card in Windows”.

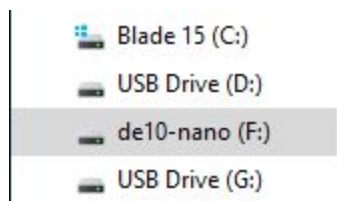
Preparing the SD Card for Windows

Windows users can follow the gist guide pretty much verbatim up until “Compiling devmem utility to interact with LUTs”. Windows does not support the commands in [bernardoaraujo's](#) guide for flashing the microSD card, so we will be downloading a 3rd party image-flashing utility called [Win32DiskImager](#) to replace those commands. Setup for Win32DiskImager is very straightforward and won't be covered in this guide.

After extracting the image from the gist and setting up the program, insert an empty and formatted microSD card into your host PC, and use it to write the image. All you need to do is set the device label to the drive letter of your microSD card, enter the image file you extracted, and click *Write*.



This process may take a couple minutes, but once it's complete, you should see that your microSD card is split into 3 partitions:

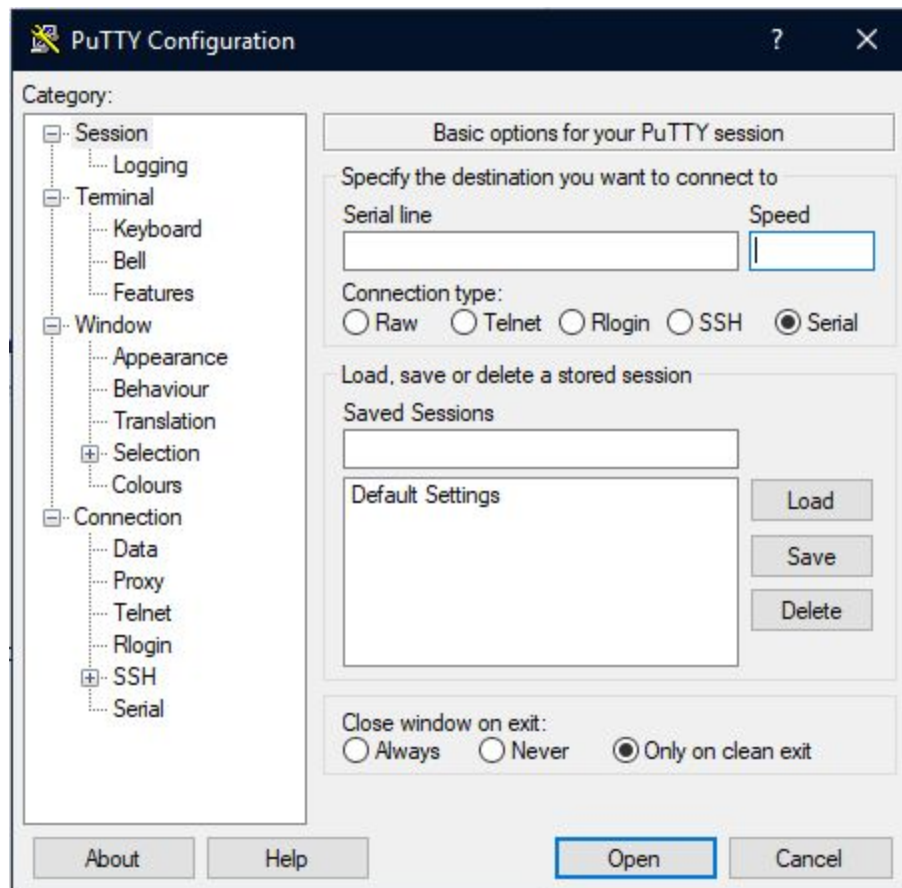


The one called *de10-nano* is the FAT32 partition, which is where we will be inserting our *soc_system.rbf* file that we generated using the batch script previously and the */FpgaServer/server* folder included in our source. You cannot access the other partitions from Windows without 3rd-party programs because they contain Linux filesystems.

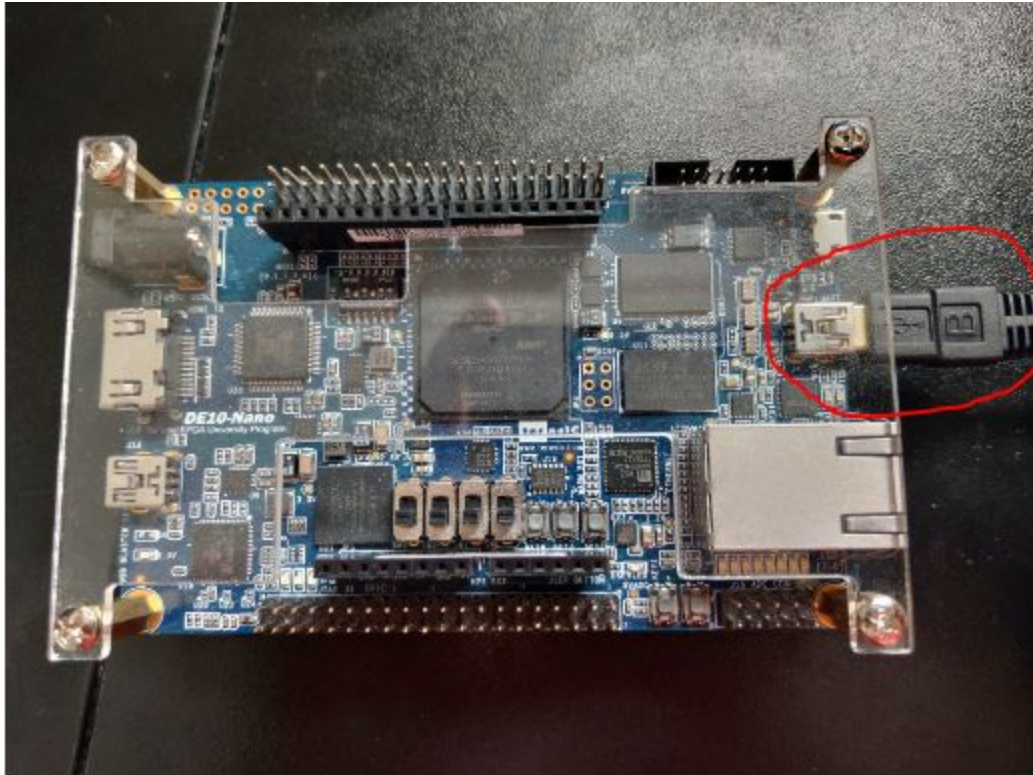
From here, you can follow [bernardoaraujo's gist.github.com guide](https://gist.github.com/bernardoaraujo) at step 3 of "Preparing the SD card" (*you can skip compiling the devmem utility and all the steps afterwards for this tutorial*). If you are familiar with establishing UART connections, the next section can be skipped.

Accessing the DE10-Nano via UART

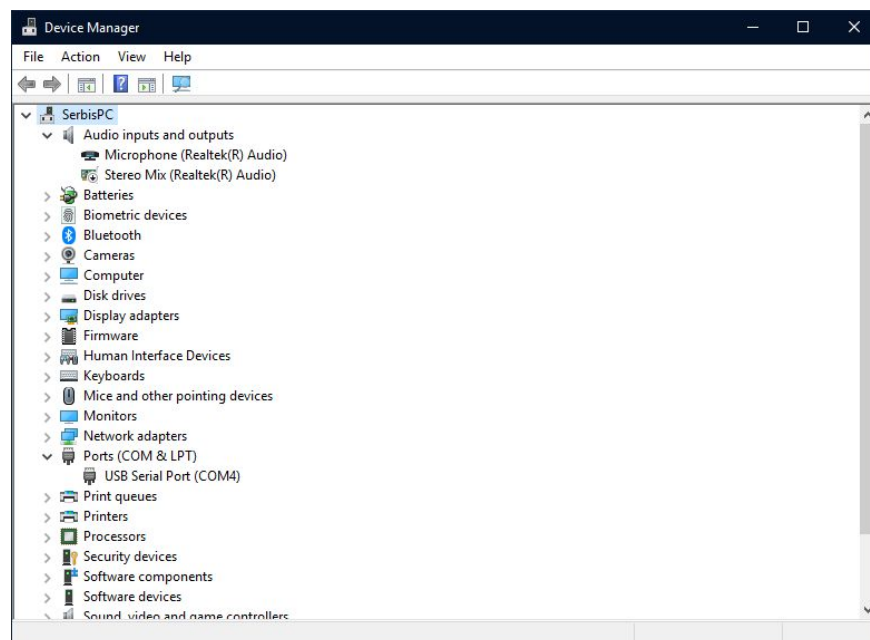
To access the terminal of the DE10-Nano, we'll need an application to UART into the board. We will be using [PuTTY](#) (version does not matter). Installation is very simple and will not be covered in this tutorial. Once you have the program installed, start it and you should see the following window:



We'll be running this at a baud rate of 115200, so we can fill the "Speed" box with "115200". The "Serial line" box is where we need to specify which serial port we're going to use. The naming convention is "COMX" for Windows and "TTYSX" for Linux, where "X" denotes the device number. To figure that out we must first plug our DE10-Nano into our laptop through the UART port so our OS can assign a device. The following picture shows where the UART-to-USB cable should be plugged into the board:



Now we need to figure out which COM port our DE10-Nano was assigned to. This is very simple in Windows (skip to the next section if you are a Linux user). You just need to navigate to the *Device Manager*, and then see which Port is active. As can be seen in the following screenshot, it was COM3 for me:



If your host PC is running Linux

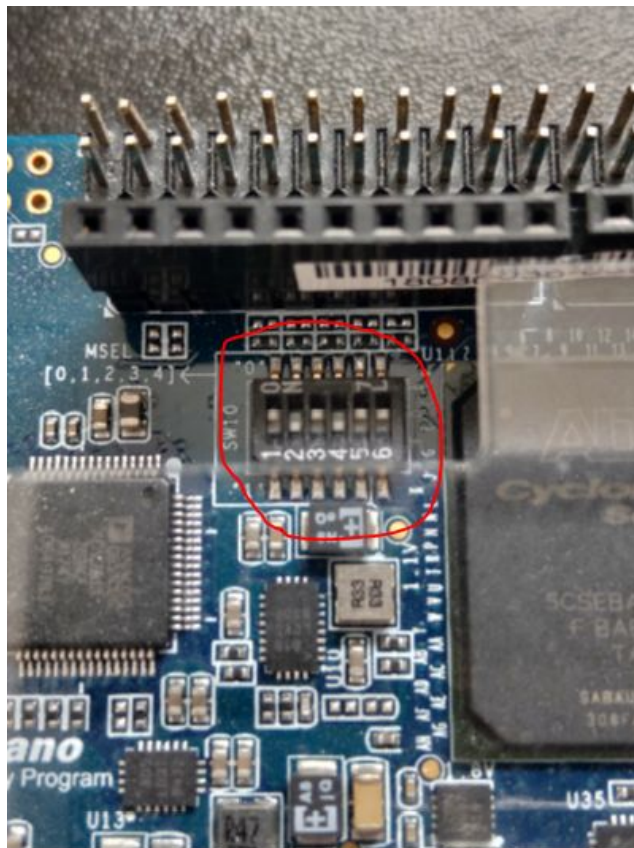
To find which serial port in Linux is being used by the DE10-Nano, type the following command:

```
→ dmesg | grep ttyS
```

This command should bring up a list of devices in use with the naming convention “TTYX”, among others. You may have to try each device to figure out which one is the actual board. An easy way to figure out which one it is, is by checking with the command, then plugging in your board, then running the command again to see what’s been added to the list.

Compiling and Running the FpgaServer on DE10-Nano

A very important step that people could have missed from the previous guide, is making sure all the MSEL switches on the board are set to the right value. These switches determine how the board is going to boot, and they need to be in the correct positions (010100). Refer to the following screenshot for what the switches look like and their intended positions:

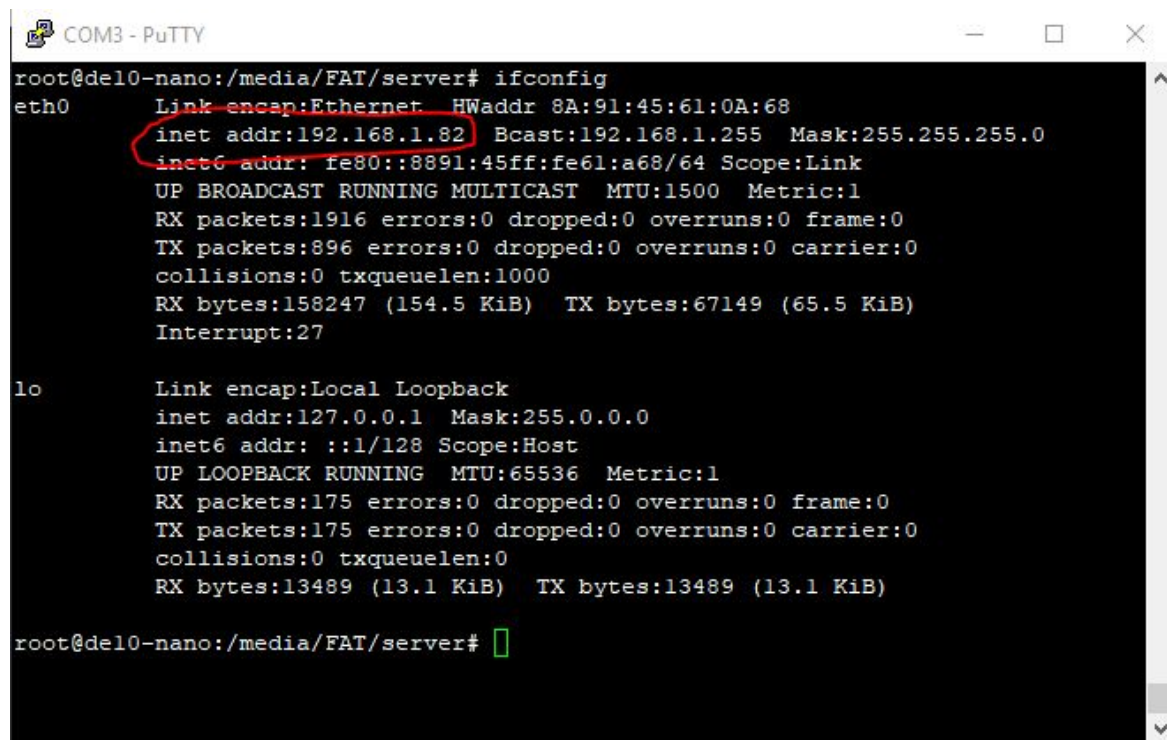


At this point, if you followed [bernardoaraujo's guide](#), you should be logged into the DE10-Nano as root. We can navigate to and view the contents of the FAT32 partition by typing the following commands:

```
→ cd /media/FAT
→ ls
```

```
root@del0-nano:~/Desktop# ls
USB_Gadget
root@del0-nano:~/Desktop# cd /media/FAT
root@del0-nano:/media/FAT# ls
LICENSE.del0-nano.rbf      del0_nano_hdmi_config.bin      server
STARTUP.BMP               dump_adv7513_edid.bin          soc_system.rbf
STARTUP.BMP.LICENSE        dump_adv7513_regs.bin          socfpga_cyclone5_del0_nano.dtb
System Volume Information  extlinux                       zImage
del0-nano.rbf              image-version-info
```

You'll see a number of files, but we're interested in the *server* folder. We are now ready to compile the FpgaServer and run on it on the board., but before we do that, we are going to want to grab the IP address of our board. To do so, type *ifconfig* in your PuTTY terminal. You should see something similar to the following screenshot:



```
COM3 - PuTTY
root@del0-nano:/media/FAT/server# ifconfig
eth0      Link encap:Ethernet  HWaddr 8A:91:45:61:0A:68
          inet addr:192.168.1.82  Bcast:192.168.1.255  Mask:255.255.255.0
          inet6 addr: fe80::8891:45ff:fe61:a68/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1916 errors:0 dropped:0 overruns:0 frame:0
          TX packets:896 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:158247 (154.5 KiB)  TX bytes:67149 (65.5 KiB)
          Interrupt:27

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:175 errors:0 dropped:0 overruns:0 frame:0
          TX packets:175 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:13489 (13.1 KiB)  TX bytes:13489 (13.1 KiB)

root@del0-nano:/media/FAT/server#
```

Take note of this IP address. Finally, to compile the `FpgaServer` program, run the following three commands:

```
→ cd server  
→ gcc FpgaServer.c Layout.c Qlut.c -o FpgaServer  
→ ./FpgaServer
```

If you'd like to make the `FpgaServer` run without it holding up the terminal, enter the following command:

```
→ ./FpgaServer &
```

Unfortunately, you won't see much happen because we have no client communicating with the server yet, so it's just sitting in its idle state. We need to head over to our host PC where we will run the Java server client.

Running the `FpgaClient` on a Host PC

To run our `FpgaClient.java` file, you must run the program in any terminal or Java development environment with the [Qupla source](#) installed (follow the README on the repo for instructions and go through building the source). We will not cover setting up a Java development environment in this tutorial.

One important step that will determine whether we can communicate with the server or not, is replacing the IP address in the `FpgaClient.java` file with the IP address we grabbed from our board. To do so, navigate to `\Qupla\src\main\java\org\iota\qupla\abra`, and change line 12 to reflect the IP address we grabbed before. In this case, my board's local IP was 192.168.1.82:

```

1  package org.iota.qupla.abra;
2
3  import java.io.BufferedOutputStream;
4  import java.io.IOException;
5  import java.io.InputStream;
6  import java.io.OutputStream;
7  import java.net.Socket;
8  import java.net.UnknownHostException;
9
10 public class FpgaClient
11 {
12     private String host = "192.168.1.82"; // old IP "192.168.1.76";
13     private InputStream inStream;
14     private OutputStream outStream;
15     private int port = 6666;
16     private Socket socket = null;
17

```

Head back to the [Qupla source repository](#) and run the following command to rebuild the source:

```
-> javac -d $HOME/Qupla/build org/iota/qupla/Qupla.java
```

With the FpgaServer running on the FPGA and the Qupla source installed and rebuilt, go into a command line and type the following command (NOTE: this command must be run in the `\Qupla\src\main\resources` directory):

```
→ java -classpath \Qupla\build org.iota.qupla.Qupla -abra -config
Qupla "sub<Tiny>(9,2) "
```

This command will generate Abra, optimize it into a config file, and load and run the config file on the FPGA. You will see “Read Failed” at the end of the server output. This just means the end of a connection, and that the hardware is back in its idle state. The result for this command should look like the following on both ends:


```
Command Prompt
QuplaSource: Qupla/print.qpl
QuplaSource: Qupla/quorum.qpl
QuplaSource: Qupla/rshift.qpl
QuplaSource: Qupla/rshiftN.qpl
QuplaSource: Qupla/types.qpl
QuplaSource: Qupla/unequal.qpl
Start dispatcher
Run Abra generator
Generate Configuration
Optimizing all_1
Optimizing equal_1
Optimizing cmp_1
Optimizing even_1
Optimizing even_3
Optimizing fullAdd_1
Optimizing fullAdd_3
Optimizing fullMul_1
Optimizing fullMulNonZero_1
Optimizing halfAdd_1
Optimizing halfAdd_3
Optimizing neg_1
Optimizing sign_1
Optimizing unequal_1
Optimizing neg_3
Optimizing neg_9
Optimizing fullAdd_9
Optimizing sub_9

Evaluate: sub<Tiny>(9, 2)
Eval: sub<Tiny>(9, 2)
==> (7) 1-1000000
Time: 26 ms
Stop dispatcher

C:\Qupla\src\main\resources>
```

```
COM3 - PuTTY
0010 indices: 000f 0011 0008, config: dee5ee67 00399de9 2088440f
Read failed

Configuration data copied successfully!
QLUT write config
0000 indices: 0000 0009 0000, config: 5e9f97a7 00397a7e 20082600
0001 indices: 0000 0009 0000, config: bf7fefdf 003efdff 20082600
0002 indices: 000a 0001 0001, config: dee799de 01ee679 2010060a
0003 indices: 000a 0001 0001, config: fff977ff 003ffba7 2010060a
0004 indices: 000b 0003 0002, config: dee799de 01ee679 20200e0b
0005 indices: 000b 0003 0002, config: fff977ff 003ffba7 20200e0b
0006 indices: 000c 0005 0003, config: dee799de 01ee679 2030160c
0007 indices: 000c 0005 0003, config: fff977ff 003ffba7 2030160c
0008 indices: 000d 0007 0004, config: dee799de 01ee679 20401e0d
0009 indices: 000d 0007 0004, config: fff977ff 003ffba7 20401e0d
000a indices: 000e 0009 0005, config: dee799de 01ee679 2050260e
000b indices: 000e 0009 0005, config: fff977ff 003ffba7 2050260e
000c indices: 000f 000b 0006, config: dee799de 01ee679 20602e0f
000d indices: 000f 000b 0006, config: fff977ff 003ffba7 20602e0f
000e indices: 0010 000d 0007, config: dee799de 01ee679 20703610
000f indices: 0010 000d 0007, config: fff977ff 003ffba7 20703610
0010 indices: 0011 000f 0008, config: dee799de 01ee679 20803e11
Read failed
```

As you can see, $9 - 2 = 7$. If we change the command to do an add with the same numbers:

→ `java -classpath \Qupla\build org.iota.qupla.Qupla -abra -config Qupla "add<Tiny>(9,2) "`

```
Command Prompt
QuplaSource: Qupla/Math/sign.qpl
QuplaSource: Qupla/Math/sub.qpl
QuplaSource: Qupla/print.qpl
QuplaSource: Qupla/quorum.qpl
QuplaSource: Qupla/rshift.qpl
QuplaSource: Qupla/rshiftN.qpl
QuplaSource: Qupla/types.qpl
QuplaSource: Qupla/unequal.qpl
Start dispatcher
Run Abra generator
Generate Configuration
Optimizing all_1
Optimizing equal_1
Optimizing cmp_1
Optimizing even_1
Optimizing even_3
Optimizing fullAdd_1
Optimizing fullAdd_3
Optimizing fullMul_1
Optimizing fullMulNonZero_1
Optimizing halfAdd_1
Optimizing halfAdd_3
Optimizing neg_1
Optimizing sign_1
Optimizing unequal_1
Optimizing fullAdd_9
Optimizing add_9

Evaluate: add<Tiny>(9, 2)
Eval: add<Tiny>(9, 2)
==> (11) -11000000
Time: 28 ms
Stop dispatcher

C:\Qupla\src\main\resources>
```

```
COM3 - PuTTY
0010 indices: 0011 000f 0008, config: dee799de 001ee679 20803e11
Read failed

Configuration data copied successfully!
QLUT write config
0000 indices: 0000 0009 0000, config: dee677b9 00277b99 20082600
0001 indices: 0000 0009 0000, config: fff9ffff 001ffff7 20082600
0002 indices: 0001 000a 0001, config: dee5ee67 00399de9 20182801
0003 indices: 0001 000a 0001, config: fffbfba7 00177ff7 20182801
0004 indices: 0003 000b 0002, config: dee5ee67 00399de9 20282c03
0005 indices: 0003 000b 0002, config: fffbfba7 00177ff7 20282c03
0006 indices: 0005 000c 0003, config: dee5ee67 00399de9 20383005
0007 indices: 0005 000c 0003, config: fffbfba7 00177ff7 20383005
0008 indices: 0007 000d 0004, config: dee5ee67 00399de9 20483407
0009 indices: 0007 000d 0004, config: fffbfba7 00177ff7 20483407
000a indices: 0009 000e 0005, config: dee5ee67 00399de9 20583809
000b indices: 0009 000e 0005, config: fffbfba7 00177ff7 20583809
000c indices: 000b 000f 0006, config: dee5ee67 00399de9 20683c0b
000d indices: 000b 000f 0006, config: fffbfba7 00177ff7 20683c0b
000e indices: 000d 0010 0007, config: dee5ee67 00399de9 2078400d
000f indices: 000d 0010 0007, config: fffbfba7 00177ff7 2078400d
0010 indices: 000f 0011 0008, config: dee5ee67 00399de9 2088440f
Read failed
```

We can see that we get $9 + 2 = 11$. You can also enter in other commands like mul\div.

Conclusion

We now have an FPGA with a working hardware design running a server that communicates directly with the Qupla emulator. That concludes this tutorial. If you have any questions, concerns, or have found any bugs, please feel free to post it in the [Iota Discord](#). This guide will be updated if any issues arise.