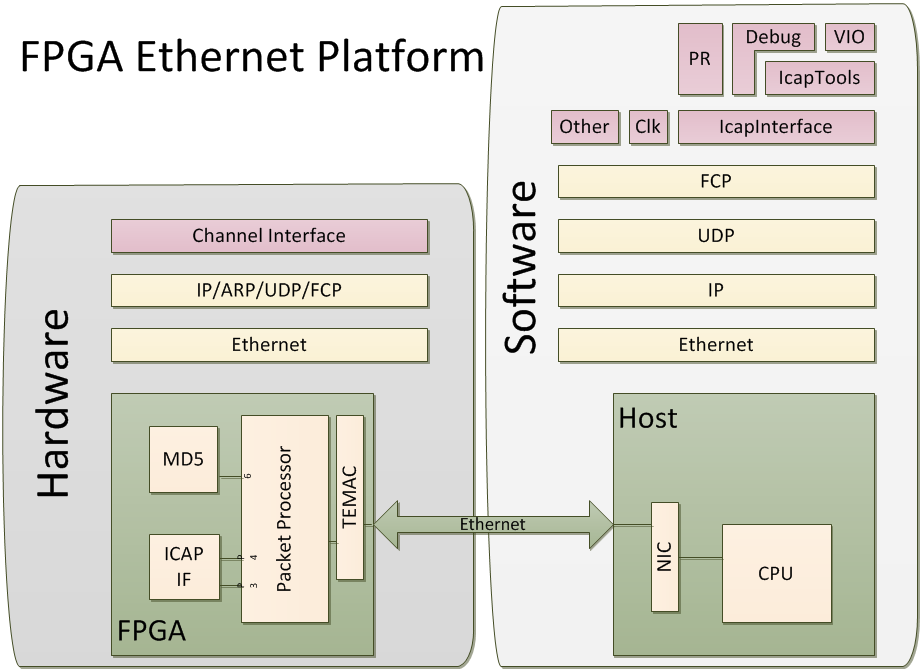
FPGA Ethernet Platform

User Guide

By Peter Andrew Thorpe Lieber  
Brigham Young University

# Introduction

The FPGA Ethernet Platform is a communications framework for FPGA to host PC communication. It uses standard network protocols to communication between an FPGA and any host computer with a network adapter and Java support. This system can be used for test vector input and output for in hardware verification. It can even be used as the main communication mechanism when appropriate. The use of standard network protocols eases the deployment process because standard networking components can be used. Having a Java based API allows many host systems to be used (Windows, OS X, Linux, etc.).

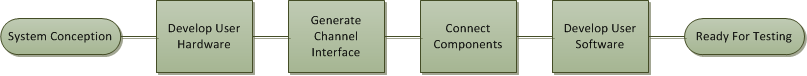
<FPGA Ethernet Platform>  


## Usage

During the FPGA development cycle, there is a point at which in hardware testing is needed. Many times, portions of the design are completed before any communication system is done. To test this, some simple, easy to use communication system needs to be temporarily used to get test vectors and sample data to the circuit. This framework can serve as that system.

# Overview

The basic flow for integrating this framework is shown in Figure 1. After developing the user hardware module(s), the channel interface is generated. This interface can present one to fifteen channels for general use. The user module(s) are then wired up to their assigned channel numbers. The hardware is now complete. The Java API, FCPProtocol, can be used to connect to and send data to/from the user module(s). The API classes are included with the user software that will generate data and/or test vectors for the hardware. Any Java program can make use of this API to connect to hardware resources.

<Figure 1: “basicflow”>  


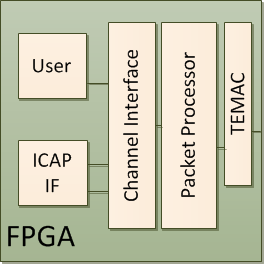
## Requirements

This framework can be used with many Xilinx FPGAs and development boards. Some boards are supported out of the box, while others, including Altera, can be supported by the user providing the missing interface circuitry, such as the Ethernet MAC wrapper. The ICAP configuration interface is only available on Xilinx chips. Table 1 shows the chips and boards supported by each main component.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Chip | Board Support | Packet Processor | EMAC Wrapper | ICAP Interface |
| Virtex 4 | ML403 | Out of Box | Out of Box | Out of Box |
| Virtex 5 | ML505/6/7, XUPV5 | Out of Box | Out of Box | Out of Box |
| Spartan 6 |  | Out of Box | User | User |
| Virtex 6 |  | Out of Box | User | User |
| Altera |  | Out of Box | User | Not Available |

## Hardware

The hardware framework consists of a packet processing circuit, Ethernet MAC wrapper, and a hardware interface. The packet processing circuit implements the entire network stack from layer 1(IP) to 3(FCP).

<Figure: hardware>  


The processor is a microcoded state machine that can be regenerated from a python description file. This file, fcpudpip.py, generates the microcode ROM file in Verilog, microcodesrc.v. This file needs to be generated by the user when the IP address needs to change. By default, it is hardcoded to 192.168.1.222. To change the IP address, first locate the IP definition at the top of fcpudpip.py.

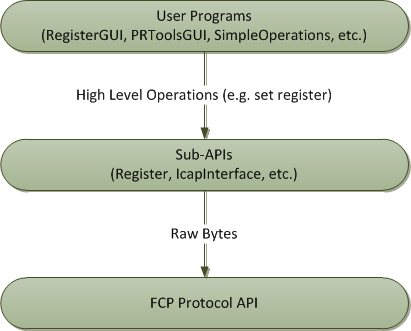
IP = [192, 168, 1, 222]

Change each number, remembering to keep the commas instead of periods. After making the change, run the assembler from the command line (python 2.6 required):

* python pasm.py fcpudpip.py > microcodesrc.v (linux)
* python pasm.py fcpudpip.py | Out-File –Encoding ascii microcodesrc.v (PowerShell)

After generating the new microcodesrc.v, replace the old one with it. Now, the FPGA will respond to the new IP address.

## Software

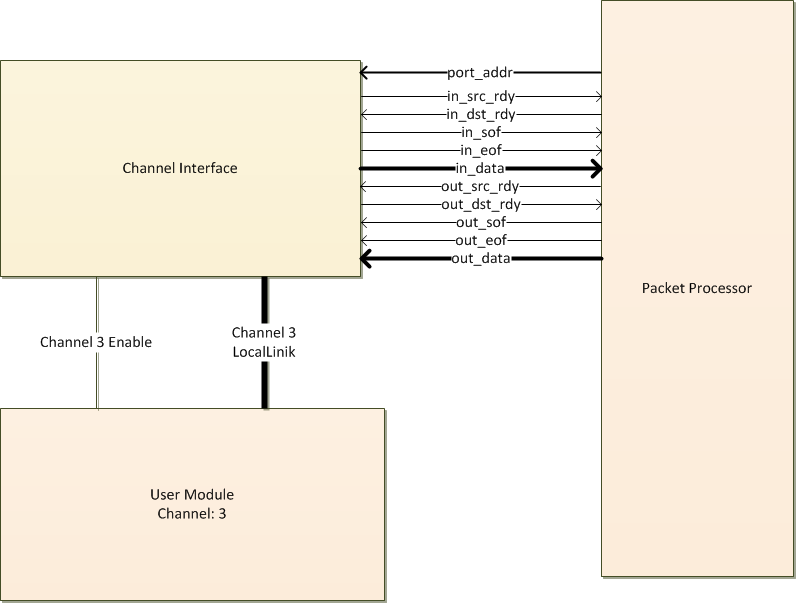
<Software API>  


The software is written entirely in Java. Because of this, the software is portable to any operating system that supports Java. The software consists of a base API, FCPProtocol, and sub-APIs to enable higher level functionality for specific modules, such as the register API in the example design (see Figure <Software API>). User programs can be written using the base API, a premade sub-API for existing modules, or a user-created sub-API. The suggested method is to first create a sub-API for your module, then use it to communicate with your hardware (see example design).

# Hardware Interface

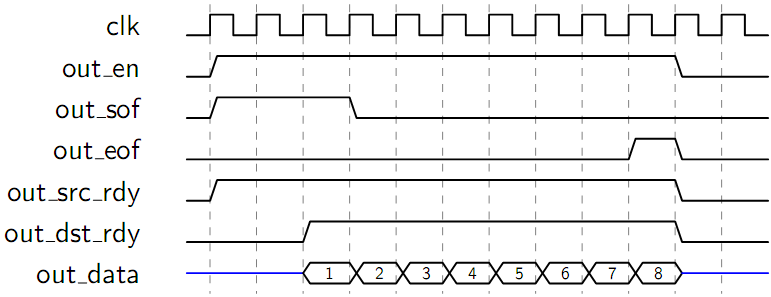
User hardware is connected to the framework via an interface based on Xilinx's LocalLink interface. The packet processor presents a channel address along with a single bidirectional LocalLink interface. The channel interface decodes the channel address to enable signals for each channel in the channel interface. So, for each channel, a LocalLink in each direction and a channel enable signal is provided. The channel interface is generated by the user so that only the needed number of channel interfaces are exposed. The tool, chifgen.py, accepts one parameter: the number of channels to expose. The output of the tool is the channel interface, channelif.v.

<Figure: Channel Interface>

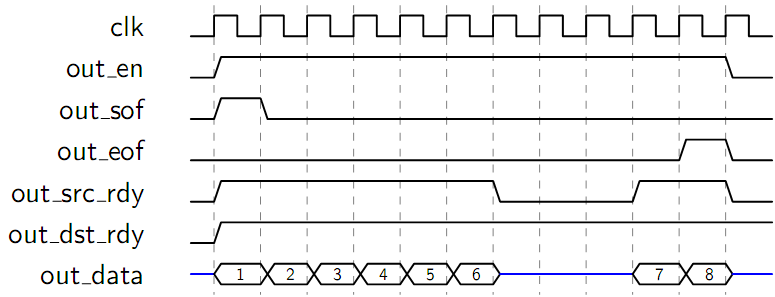


Each channel gets an in and out LocalLink interface. The LocalLink interface is specified in Xilinx App Note XAPP691. It is repeated here for convenience. There is one design consideration that should be followed when using the channel interface. The control signals generated by the user module (out\_dst\_rdy, in\_src\_rdy, etc.) should not be derived from control signals of the channel interface without registering them first. In other words, these signals should be Moore outputs of the user state machine. This will prevent timing violations.

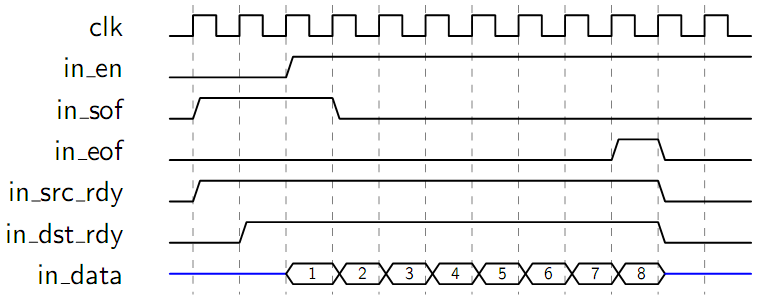
Data is transferred from and to the channel interface via an 8 bit input and output bus. Each direction has four control signals governing the operation. The out\_src\_rdy and out\_dst\_rdy signals control when data is valid and data is accepted, respectively. Only when both signals are asserted data is considered transferred. The out\_sof and out\_eof signals are asserted on the first and last byte, respectively, of each data packet. These signals are often used to trigger user designs to begin. The enable signal is a global enable for that channel. For data to be valid coming from the channel interface, en and out\_src\_rdy must both be asserted. Figure <outdata> shows an eight byte sequence being sent to the user logic.

<Figure: Data Output Timing Diagram>  


As seen above, the user logic generated signal, out\_dst\_rdy, gets asserted when it is ready for data. At that point, the eight bytes of data are transferred to the user logic. If at any time the out\_src\_rdy signal is deasserted, the data is not valid and the user logic must wait for the signal to continue receiving data. Figure <outdataflow> shows an example where the user logic is expecting 8 bytes, but must wait midstream for the last two.

<Figure: Data Output Timing Diagram with Flow Control>  


For data transfers in the other direction, from user logic to channel interface the procedure is identical. Figure <indata> shows an example of this. Flow control can also cause pauses in communication similar to Figure <outdataflow>.

<Figure: Data Input Timing Diagram>  


# Software API

All capabilities of the FPGA Ethernet Framework are encompassed in a single API, FCPProtocol.java. Five basic functions support most operations:

1. void connect(InetAddress address);
2. void sendData(int channel, ArrayList<Byte> data, int numBytes);
3. void sendDataRequest(int channel, int numBytes);
4. ArrayList<Byte> getDataResponse();
5. Void disconnect();

To connect to the FPGA, the correct IP address is needed. An FCPProtocol object is created, then the connect function called. Before any data transfer functions can be called, the isConnected() function should return true. Below is an example of the procedure for connecting.

FCPProtocol protocol = new FCPProtocol();  
protocol.connect(InetAddress.getByName(“192.168.1.222”));  
while (!protocol.isConnected());

Sending data consists of a single function call. The data, an ArrayList or array of bytes is passed along with the channel number and number of bytes to send. The below code would result in a waveform on the channel interface similar to Figure <Data Output Timing Diagram>.

ArrayList<Byte> data = new ArrayList<Byte>();

for (int i=0; i<8; i++) {

data.add(new Byte((byte)(i+1)));

}

protocol.send(1, data, 8);

Receiving data is split into two function calls, a data request and getting the response. The data request sends a request packet to the FPGA, which in turn responds with the data. The data request function is non-blocking. The FCPProtocol receives the response some time later. This response is retrieved with the getDataResponse() function. This function blocks until a response is in the receive buffer, then it returns an array of bytes containing the data sent from the FPGA. The responses are given by getDataResponse() in the same order that the requests were sent.

Byte[] response;

protocol.sendDataRequest(1, 8);

response = protocol.getDataResponse();

When the program exits, the disconnect function should be called to terminate the send and receive tasks.

protocol.disconnect();

For a more detailed description of the Java API, please refer to the Javadoc documentation included with the package.

# Framework File Descriptions

This section contains descriptions of the files provided by the framework within their folder structure. Bolded items are folders, the others are files.

* **doc**
  + FPGA Ethernet Platform User Guide.pdf: this document
* **hdl: hardware source files**
  + **boardsupport**
    - ml403.ucf: Constraints file for the ML403 development board
    - xupv5.ucf: Constraints file for the XUPV5 development board
  + **chipsupport**
    - v4: Virtex 4 Xilinx TEMAC wrapper files
      * dcm\_reset.v
      * emac0\_fcs\_blk\_mii.v
      * eth\_fifo\_8.v
      * mii\_if.v
      * rx\_client\_fifo\_8.v
      * sync\_block.v
      * tx\_client\_fifo\_8.v
      * v4\_emac\_v4\_8.v
      * v4\_emac\_v4\_8\_block.v
      * v4\_emac\_v4\_8\_locallink.v
    - **v5: Virtex 5 Xilinx TEMAC wrapper files**
      * eth\_fifo\_8.v
      * gtp\_dual\_1000X.v
      * rocketio\_wrapper\_gtp.v
      * rocketio\_wrapper\_gtp\_tile.v
      * rx\_client\_fifo\_8.v
      * rx\_elastic\_buffer.v
      * tx\_client\_fifo\_8.v
      * v5\_emac\_v1\_6.v
      * v5\_emac\_v1\_6\_block.v
      * v5\_emac\_v1\_6\_locallink.v
    - enetplatformv4.v: Virtex 4 Ethernet Platform
    - enetplatformv5.v: Virtex 5 Ethernet Platform
  + **modules: modules that communicate over FCP**
    - **md5**
      * md5.v: implementation of the md5 hash sum
      * port\_md5.v: channel interface for the md5 module
    - **port\_icap**
      * port\_icap\_buf.v: channel interface for the icap
      * shiftr\_bram.v: FIFO using a BRAM
    - **port\_register**
      * port\_register.v: 32 bit register
  + **packetprocessor**
    - alunit.v: ALU Unit
    - checksum.v: 16 bit Checksum Unit
    - comparelogic.v: compare logic for the ALU
    - gensrl.v: generic SRL, will infer an SRL on most Xilinx parts
    - lpm\_mux2.v: parameterizable 2 input mux
    - lpm\_mux4.v: parameterizable 4 input mux
    - lpm\_mux8.v: parameterizable 8 input mux
    - lpm\_stopar.v: 8 to 16 bit shift register
    - microcodelogic.v: horizontal microcode logic for packet processor
    - microcodesrc.v: source for the horizontal microcode logic
    - patlpp.v: packet processor
    - regfile.v: 16 register file
    - shiftr.v: FIFO from SRLs
  + **tools**
    - **ChannelInterfaceGenerator**
      * chifgen.py: Generates the channelif.v for 1 to 15 channels
    - **PythonAssembler**
      * fcpudpip.py: implementation of FCP/UPD/IP stack in the packet processor
      * pasm.py: assembler for embedded packet processor language
      * patlpp.py: definitions for embedded packet processor language
  + **toplevel**
    - topv4.v: Virtex 4 basic top level
    - topv5.v: Virtex 5 basic top level
    - topv5\_md5.v: Virtex 5 top level with MD5 module
    - topv5\_simple.v: Simple example design with no ICAP
* **java: software source files**
  + **doc: javadoc**
  + **examples: example uses for the API**
    - FCPInterface.java: Send bytes to and from FCP channels
    - MD5GUI.java: Calculate an MD5 hash on the FPGA
    - PRToolsGUI.java: Send partial reconfiguration bit files to the ICAP
    - Simple.java: Interface to the LED, DIP, and Port Register module
    - SimpleOperations.java: Simple reading and writing from a single channel
    - ThroughputTest.java: Test the throughput of the connection
  + **fcp: Implementation of the FCP protocol and base FCP API**
    - FCPException.java
    - FCPPacket.java
    - FCPProtocol.java: Base user API for FPGA Communication
    - FCPReceiveThread.java
    - FCPSendThread.java
  + **SubAPI: Sub-API’s built atop FCPProtocol**
    - IcapInterface.java: Icap API
    - SimpleInterface.java: LED, DIP, and Register API
  + **util: Utility functions**
    - StringUtil.java
* **example: Example design files**
  + simple.prj: Project file for the simple example
  + simple.ucf: Constraint file for the example

# Example Design

Included with this framework is a complete and simple example design. The design targets an XUPV5 development board. It exposes two FCP channels in the hardware. When written to, channel 1 sets the 8 LEDs to the binary representation of the byte written to the channel. When read from, channel 1 reads the 8 DIP switches. Channel 2 is connected to a 32 bit register module. This module allows a 32 bit value to be read and written in little endian order. The hardware is accompanied by an API for the 32 register and a command line program to read and write the register, LEDs, and DIP switches.

## Hardware

The top level module, top\_simple.v, instantiates the Ethernet platform, the channel interface, and the register module. It also includes code for controlling the DIP switches and LEDs. The code for channel 1 operation of the DIP switches and LEDs is shown below:

reg [7:0] DIP\_r;

reg [7:0] LEDr;

wire [7:0] LEDnext;

assign LEDS = LEDr; // Assign the LED register to the LED pins of the FPGA

always @(posedge clk\_local)

begin

DIP\_r <= DIP; // Register the DIP switch contents

end

always @(posedge clk\_local)

begin

if (rst\_local)

LEDr <= 0;

else if (ch1\_wen & ch1\_out\_src\_rdy) //If channel 1 is enabled & source is ready

LEDr <= LEDnext; // Write the next value to the LED register

end

assign ch1\_in\_sof = 1; // Only one byte is ever read at a time from channel 1,

assign ch1\_in\_eof = 1; // so, both start and end of frame are always asserted.

assign ch1\_in\_src\_rdy = 1; // Always ready with DIP switch value

assign ch1\_out\_dst\_rdy = 1; // Always ready to receive writes to LEDs

assign ch1\_in\_data = DIP\_r; // Connect registered DIP to input data of channel 1.

assign LEDnext = ch1\_out\_data; // Connect output of channel one to next LED value

The LED and DIP switch functions for channel 1 are simple. Since only one byte is read or written, the hardware can always be ready to send or receive. Therefore, the control signals are all tied high. The start and end of frame signals are tied high because all reads are only one byte long. For writing to the LEDs, whenever the source is ready, the new value is clocked into a register. This register is directly wired to the pins for the LEDs. For reading the DIP switches, the value from the pins are clocked once, then tied directly to the data port.

Channel two is used to read and write a 32 bit register. To illustrate the suggested flow, the module, port\_register.v, implements this functionality. It presents the correct interface signals to be connected directly to channel 2 of the channel interface.

module port\_register(

input clk,

input rst,

input wen,

input ren,

input in\_sof,

input in\_eof,

input in\_src\_rdy,

output in\_dst\_rdy,

input [7:0] in\_data,

output reg out\_sof,

output reg out\_eof,

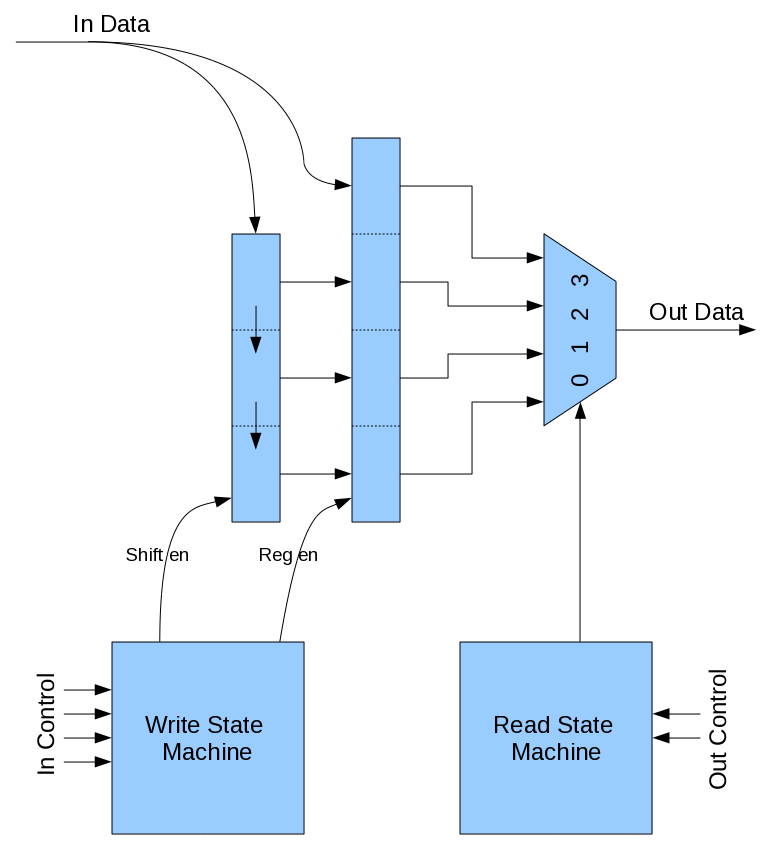
input out\_dst\_rdy,

output out\_src\_rdy,

output reg [7:0] out\_data

);

To write the register, 4 bytes are sent over channel two. A simple state machine shifts the first three bytes in, then on the forth bytes, it writes the whole word to the register to avoid invalid values (see Figure Register). Another state machine controls the reading. It reads each value directly from the register, and selects which byte with a mux using the state as the select line.

<Figure: Register>  


## Software

On the software side, a simple API wraps the low level calls to the FCP layer in four functions:

* byte getDIP()
* void setLED(byte value)
* int getRegister()
* void setRegister(int value)

The first two functions read and write the DIP switches and LEDs, respectively. The getRegister function read 4 bytes from channel 2, then assembles them into a 32 bit integer, in little endian order. Its implementation is shown below.

public int getRegister() {

protocol.sendDataRequest(2, 4);

byte[] bytes = protocol.getDataResponse();

int res = (((int)bytes[3] & 0xff) << 24) | (((int)bytes[2] & 0xff) << 16) |

(((int)bytes[1] & 0xff) << 8) | (((int)bytes[0] & 0xff));

return res;

}

The setRegister function sends 4 bytes in little endian order from a 32 bit integer. It must first mask and shift each byte of the given integer, and assemble it into an array. Then, it simply calls the send function of the FCP protocol.

public void setRegister(int value) {

ArrayList<Byte> bytes = new ArrayList<Byte>();

bytes.add(new Byte((byte) (value & 0xff)));

bytes.add(new Byte((byte) ((value >> 8) & 0xff)));

bytes.add(new Byte((byte) ((value >> 16) & 0xff)));

bytes.add(new Byte((byte) ((value >> 24) & 0xff)));

protocol.sendData(2, bytes);

}

These four functions are then used by the example command line program to allow a user to read the DIP switch values, change the LED states, and read and write the 32 bit register. There are four commands:

* w <integer>: right the integer to the register
* r: read the value of the register
* l <byte>: set the LEDs to byte
* d: get the DIP switch positions

An example transcript of the command line interface is shown below.

* Received Connection Ack
* Connected to: 192.168.1.222 on port 12289
* Welcome to the Example Design
* r
* Register Value: 0
* w 4923
* r
* Register Value: 4923
* d
* DIP Switch Value: 27
* d
* DIP Switch Value: -39
* l 185
* r
* Register Value: 4923
* quit
* Goodbye!