**Echo Server Script**

**Description**

The LWIP echo server example (which is run on the ARM processor) has been utilized to implement a simple echo server for communication between the PC and FPGA over Ethernet. The objective of this C program is to receive data from the PC and transmit to the FPGA fabric and vise-versa. The example is a template that is provided by the Vivado SDK, so it can be set up with ease. A Python script sends image data and acts as a client that communicates with the FPGA that acts as a server. The echo server implements a simple server which at the default mod echoes the inputs to the output. For our application, since we need to send and receive different values, we had to change the default functions and settings to fit our application. All the changes are in the echo.c file within the recv\_callback function. The implemented code is still dependent on the basic flow of the LWIP echo server which means that for any received packet, there should be a response (echo) from the server to the client. So, sending a packet and not receiving the echo in a period of time, will result in an error in the current application and causes the connection to time-out. It means that if we send a frame from PC to the FPGA (because of using echo server) we should receive and echo back to the PC, but it does not have to be the same size of the received packet.

To explain the modifications that have been made on the echo.c file, we start by explaining the flow of information between the PC and the FPGA. The echo.c file is commented in detail to provide a more in-depth explanation as well. At the first step, the user-defined parameters are sent from the PC to the echo server as the first packet. So, in the recv\_callback () function inline “if(packet\_counter == 0)”, it iterates through the whole received packet and writes the parameters into the BRAM. Every single parameter (image height, image width, …) are stored in 10 digits. Within this statement, the received values over ethernet are in string format (required via Python). A function defined at the top of the file called “atoi” converts the values received from a string to an integer so that they can be written into BRAM. The parameter data, for ease of use at the application level, is written three separate times to distinct BRAMs. The if statement then iterates through all the parameters in the received packet until it reaches the end. At the end of processing the parameters packet, a slave register which represents the params\_done signal is set to “1” to tell the other IPs the parameters are ready in BRAMs.

The next packets received after the parameter data should be integer values from the image/frame data. The max size for data in a TCP packet is 1460 bytes, however, due to errors and issues we experienced we lowered that number to a max of 1344 bytes. This number is useful to us for the following reasons. When we pull a single pixel value from the image it can have a size of one digit, two digits, or three digits with a max value begin 256 and a min value of 0 (I.e.: 4, 56, 256). A single image can contain up to 415,744 pixels which are a lot of data to convert from a string to an integer on the echo server. The images we use in our example are of size 448x232 which equates to 103,936 pixels per frame. Because the pixel values vary between 0-256, we needed a way to separate the pixels without using commas (searching a string and removing a portion of the string took too long on the echo.c server and caused a timeout). So before the pixel values are sent over ethernet from the PC to the FPGA, they are scanned for their length and ‘0’s are appended to the front of the number. So if a pixel value is equal to ‘1’, the value sent the FPGA would be “001”; if the value was “36” the new value sent the FPGA was “036”. This allowed us to separate each pixel on the echo.c server with ease by knowing every 3rd number is the end of a pixel. When the string is converted to an integer, the leading zeros disappear. This enabled us to add a simple loop of “if(i%3 == 0)” that was able to separate the individual pixel values. The max packet size is 1344, and if every pixel has 3 values associated with it, then each packet can send a total of 448 pixels of data. This also means that for an image size of 103,936, a total of 232 perfectly even packets of image data are sent to the FPGA.

Once the pixels are received on the echo.c server, they are converted from a string to an integer using the “atoi” function mentioned earlier. After that, they are converted from an integer to a float data type. Once that is finished, the numbers are normalized by dividing the pixel value by “256” so that whatever the pixel value is will be a decimal between 0 and 1. Next, that normalized number is sent to a function called “Dec\_to\_IEEE” that converts the decimal to IEEE 754 single-precision floating-point 32-bit binary format. During this process, the number is converted back into a string so the 32-bit binary can be accurately represented; this can’t be done using the int data type. This “number” is stored in a char\* array that is then converted to an integer. This part is a little strange, but it is a conversion trick. The binary number that is in string form is the normalized number we need to write into BRAM for the correlation to be performed on, however, we cannot write strings to BRAM. As stated earlier, we can’t save a binary number into the int data type either because it won’t accurately represent the number. However, the 32-bit binary number in string format is converted to an integer that represents a totally different number than the original one. For example, the number “0.5” in IEEE 754 binary is represented as “00111111000000000000000000000000”. When this number is converted to an integer it represents “1,056,964,608”; this is were the conversion trick comes into play. When the integer “1,056,964,608” is written into BRAM, it automatically is represented in binary format which was the original value we started with. Once all numbers are written into BRAM, an if statement is used to count the number of pixels that have been received to determine which frame we are on. By keeping track of the frame number in the echo.c server, we are sending fewer packets and data to the server and the server also writes to a slave register in the GradientsMulti IP that keeps track of the number so it can be used in the application. Lastly, each packet is echoed back to the PC to avoid a timeout on the connection.

At the end of each frame that is sent, a dummy packet of all zeroes, length 15 (“000000000000000”) is sent to the echo server that indicates a full-frame has been sent and received and then the address changes to which BRAM is being written to. Two BRAMs exist to write the original images to and they alternate between holding the “reference” frame and the “deformed” frame. After the first two frames have been sent from the PC to the echo server, the Python script will loop through a function that sends the packet “Wait” which has a length of four. When this length of packet is received, it will read from a slave register in the Gamma IP known as “process\_done” that indicates to the echo server that the first correlation on the frames is finished. If it is not finished, the echo server responds with the packet “Wait”. This will continue until a ‘1’ is written into the Gamma IPs slave register. Once a ‘1’ is written that indicates the correlation is finished, the echo.c server will write a ‘1’ into another slave register in the Gamma IP known as “new\_frame” that indicates Gamma a new frame will be sent to it soon and it needs to restart. Then a “Send” packet is sent back to the PC where the Python script will send the next frame in line. Lastly, when all of the frames have been sent over to the FPGA, the Python script will send a packet that reads “Done! so that the echo server knows there will be no more images coming. The nested if statement checks to see if the slave register “process\_done” has a ‘1’ written into it in the Results IP that lets the echo server know the correlation is finished and the results are ready. If it is not ready, the echo server will echo back to the Python server “Done!” in a loop until the results are done. When the results are done, the echo server will send the packet “Okay!” back to the Python script to let it know that all results are ready. The Python script then sends over a packet that says “results” and the echo server will read from the BRAM that all the results are saved to that is connected to the Results IP. The results are comma-separated and broken up on the Python script side to be formatted and written into final text files. Currently, the results only concatenate the three values that are produced from the application which are: Displacement\_X, Displacement\_Y, and Rotation\_Z. Lastly, the echo server will respond with “finished” that lets the Python script know that all results are finished and can end the correlation. The remainder of the LWIP echo server is automatically generated when the template is selected. It is to be noted that we keep all BSP settings as their default except for the tcp\_mss which is set to a max packet size of 1344.