MicroBlaze TFTP Server User Guide

Lorne Applebaum
appleba@eecg.utoronto.ca

August 25, 2004

1 Preamble

This document describes the intended method of use for the MicroBlaze TFTP Server. For detailed information regarding how the server is implemented, see the commented source code.

The server was designed using Xilinx's EDK v6.2.2. It was tested using Xilinx's Virtex II Multimedia FF896 Development Board. The TFTP client used for testing was the tftp client included in GNU's inetutils v1.4.2.

The purpose in mind when designing the server was for the server to be a means to get data from a host computer onto an FPGA design. Thus, no attempt was made to create or use a filesystem. The server contains entities known as "files", however, the files are not intended be created or destroyed during the server's operation. Rather, the files should be thought of as portals for a data path.

2 Features

- Compliant with the RFC 1350 TFTP Protocol definition
- Capable of handling multiple concurrent connections
- Implemented using LWIP
- Interaction with user software using function callbacks

3 Limitations

Limited mode support The TFTP protocol is defined to have 2 modes: binary and ascii. Since the MicroBlaze has no defined standard to storing ASCII text, an ascii mode would be meaningless. The TFTP server ignores the mode field of TFTP transfer requests and treats all transfers as raw binary transfers.

No Error Packet Handling The TFTP server handles all errors based on timeouts. Received error packets are ignored. Error packets are considered a courtesy in the TFTP protocol rather than a necessity. However, the server will send out error packets when it encounters an error.

Large Code Size At the document's time of writing, the code for the TFTP server and supporting libraries was too large to fit on a Virtex II's internal block RAM. Thus the TFTP server is required to run off external memory.

4 The Server's Files

The server is contained within two files: `tftp.c` and `tftp.h`

`tftp.c` Contains the function definitions used by the TFTP server.

`tftp.h` Contains the function declarations and parameter definitions. Some parameter definitions can be altered by the user.

Using the server requires including these two files in an application. It may be more useful to think of `tftp.c` and `tftp.h` as a library used by a user written server application.

5 The Example TFTP Serving Application

Included with the TFTP server code is an example application. Its code can be found in `examplemain.c`. The example application instantiates a server with two "files". The first file writes to and reads from a section on an external ZBT bank. The second file is write only and writes transfered data to the MicroBlaze's standard output.

5.1 Getting the Example Application Running

The following section contains some instructions that are specific to the Multimedia Development Board mentioned in the Preamble. However, with minor changes, the sample application should be able to run on other boards.

5.1.1 Creating the System

1. Using Xilinx Platform Studio, create a new project with Base System Builder. Include within the system, the XMD debugging interface, the RS232 IO device, the Ethernet_MAC and the ZBT_512Kx32 devices. The default settings for these devices will suffice.
2. Select the menu item Project→Add/Edit Cores... (dialog). From the Peripherals tab, add an instance of the opb_timer IP and give at an available address range (0xffff0000 to 0xffff00ff works). The instance name of the timer should be opb_timer_0.
3. From the Bus Connections tab, add the timer to the OPB bus.
4. From the Ports tab, add the INTERRUPT port of the MicroBlaze. Add the OPB_Clk and the Interrupt ports of the timer. All of the ports should be internal in scope. Connect the timer's OPB_Clk to sys_clk_s. Connect the timer's Interrupt port to the INTERRUPT port of the MicroBlaze by setting the Net Name associated with the Interrupt port to microblaze_0_INTERRUPT. Clock OK.
5. Select the menu item Project→Software Platform Settings.... Switch to the Processor and Driver Parameters tab. In the Driver Parameters section, enter tmrInterruptHandler as the interrupt_handler for the timer. The function entered will be within `examplemain.c`.

5.1.2 Setting up LWIP

6. Switch to the Software Platform tab. In the Libraries section of the Software Platform tab, click the check box next to lwip.

   Switch to the Library/OS Parameters tab and click on the ... button next to the emac_instances for lwip. A dialog box will pop up. Click the Add. Under the emac_instname column enter Ethernet_MAC. This is the instance of the EMAC created by Base System Builder. Under the remaining columns enter an arbitrary MAC address. Click the OK button of the dialog box.

   Click the OK button of the Software Platform Settings screen.
7. Select the menu item Tools→Generate Libraries and BSPs. This will copy several files into the system's project directory and then compile the libraries. By default, Xilinx has LWIP enabled for TCP/IP communication. Since TFTP runs on UDP/IP communication, a few settings must be changed.

8. Open the `microblaze_0/libsrc/lwip_v1_00_a/src/contrib/ports/v2pro/lwipopts.h` file in the system's project directory. Alter the following definitions:

```
MEM_SIZE                600
MEMP_NUM_PBUF           0
MEMP_NUM_UDP_PCB        8
MEMP_NUM_TCP_PCB        0
MEMP_NUM_TCP_PCB_LISTEN 0
MEMP_NUM_TCP_SEC        0
PBUF_POOL_SIZE          10
PBUF_POOL_BUFSIZE       600
LWIP_TCP                0
LWIP_UDP                1
```

   For an explanation of the meanings of some of these definitions, see section 10.1.

9. Select the menu item Tools→Xygwin Shell and type:

```
cd microblaze_0/libsrc/lwip_v1_00_a/src
make all
```

   This will regenerate `microblaze_0/lib/liblwip4.a` with the appropriate options. Copy the altered `lwipopts.h` to a backup directory since the file will be overwritten if the libraries are regenerated.

5.1.3 Adding the Software to the System

10. Select the menu item Project→Add Software Application Project... and enter exampletftp as the Project Name.

11. Right click on Project: exampletftp and disable the Mark to Initialize BRAMs option. Double click on Sources and add the three software files: `examplemain.c`, `tftp.c` and `tftp.h`.

12. Double click on Compiler Options. Set the Program Start Address to 0x80600000 so that the application will run on the external ZBT memory. In the Directories tab, within the Linker Options, set Libs to link (-l) to lwip4. To have the server to print extra debugging information to the standard output, switch to the Advanced tab and set the Program Sources Compiler Options to -DTFTP_DEBUG. Click OK.

13. Right click on Project: exampletftp and select Build Project. Errors such as "undefined reference to 'udp_remove'" are indicative of the lwip library being overwritten by the default version. The `liblwip4.a` library will have to be regenerated using the backup version of `lwipopts.h` and following the above instructions.

5.1.4 Running the Server

14. Download the system to the FPGA by selecting the menu item Tools→Download.

15. Select the menu item Tools→XMD. Type the following:

```
mbconnect stub
dow exampletftp/executable.elf
run
```

   A few messages should be output to the MicroBlaze's standard output indicating that the server is running.

5.1.5 Interacting with the Server

16. The server has two files that are able to be interacted with: "zbt" and "stdout". The tftp server is set to the IP address 192.0.0.2. Use a tftp client to connect to this address.

17. Get the "zbt" file from the server in binary mode. It will create a file on the client machine. The file contains the contents of a part of the external ZBT memory. At this point it is probably random data. Edit the file, being careful not to change the size (The server will respond with an error if an attempt to write outside the bounds of the ZBT is made). Put the edited "zbt" file back on the server. Rename the local version and get the file off the server once again (don't forget to use binary mode). Compare the two files and check that they are in fact identical.

18. Rename a text file on the client machine to "stdout". Upload this file to the server. The contents of the file should be output on the MicroBlaze's standard output.

6 The Concept of Files on the TFTP Server

As mentioned in the preamble, the concept of files should be thought of as a data path to get data onto the FPGA. Files are not created if they do not already exist. There is nothing resembling an operating system's filesystem.

The files on the server contain three pieces of information: a filename, a read call-back function, and a write call-back function.

The filename is used to match a read or write request of a client with a file. The call-back functions are the means by which the user of the server gets data from a client to server transfer (a "put" calls the write call-back) or supplies data for a server to client transfer (a "get" calls the read call-back).

Files are supplied to the server during the MicroBlaze's runtime. Files are supplied to the server using the `registerFile()` function. It takes three parameters: the filename and the two call-back functions. Either of the two call-back functions can be `NULL` if the file is intended to be either read-only or write-only. Supplying files is intended to be done before the server is put in operation by a `runTftpServ()` call.

7 The Call-Back Scheme

In order to maintain a small overhead, some parts of the TFTP protocol and the LWIP library are apparent in the server's call-back scheme.

7.1 TFTP and the Call-Back Scheme

The TFTP protocol defines that data be sent in 512 byte packets. The user specified call-backs are called each time a packet is received for the call-back's associated file or each time a packet is required to be sent. The TFTP protocol also defines that packets of size less than 512 bytes should be recognized as a transfer terminating packet. Thus, the reading call-back used when sending a file to a client should supply 512 bytes of data on every call unless it is terminating the transfer. Making the 512 byte packet size apparent in the call-back scheme saves the requirement to buffer an entire file before sending it to the user.

Also apparent in the call-back scheme is the TFTP protocol's concept of block numbers. Each data packet sent has an associated block number. The block numbers start at 1 for the first packet and increment by one for each subsequent packet. The block number is passed to the user supplied call-back and should be used to determine what data to send. The use of the block numbers is most important for the reading call-back since it may be called more than once for a given block number if the client failed to receive the last transfer attempt. The writing call-back can expect to get packets in simple succession as long as multiple connections are not accessing the file.

7.2 LWIP and the Call-Back Scheme

LWIP using a special buffering scheme. It keeps data in structures known as pbufs. Using these structures, data is stored in a linked list, as depicted in Figure 1. For both read and write call-backs, a pbuf linked list is supplied. The list must be traversed to be filled or read. The pbuf data members that are significant to the call-backs are listed below.

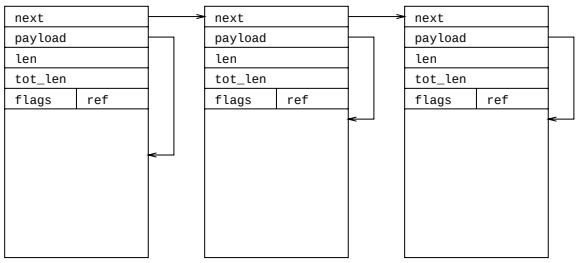

Figure 1: A linked list of pbuf structures (Taken from LWIP documentation).

next A pointer to the next pbuf structure in the linked list. Has `NULL` value if the current structure is the last node on the list.

payload A pointer to the data associated with the current node in the linked list.

len The size in bytes of the data associated with the current node in the linked list.

tot_len The size in bytes of the data contained within the entire linked list.

7.3 The Call-Back Function Details

The call-back functions are required to have the form:

```
int tftpCallBackFunc(USHORT blockNo, struct pbuf * buf);
```

The `blockNo` parameter contains the block number of the current packet. The `buf` parameter is a pointer to the first pbuf structure in the linked list containing the data.

The call-back function is required to return the number of bytes it has read from the pbuf linked list or written to the pbuf linked list. For all but the last packet in a transfer, this should be 512.

For portability, the TFTP 512 byte packet size has been defined with `TFTP_PACKET_SIZE`. This definition should also be used in the code of the call-backs.

8 The TFTP Server's Timing Mechanisms

The TFTP protocol is highly dependent on timeouts for determining errors. Since the MicroBlaze does not have an internal timer, timing must be supplied externally. The TFTP server is supplied its timing capabilities via the `tftpTimeoutTicker()`. The `tftpTimeoutTicker()` function takes no arguments and must be called on a regular interval. In the example application this is accomplished using an external timer which is used to interrupt the MicroBlaze. The `tftpTimeoutTicker()` function is called from the interrupt handler (the interrupt service routine).

The duration of time considered a timeout is determined by parameters defined in the `tftp.h` file.

The `TFTP_MAX_TIMEOUT_TICKS` parameter is used to indicate after how many `tftpTimeoutTicker()` calls an interval is considered a timeout.

The `TFTP_MAX_TIMEOUTS` parameter is used to indicate after how many consecutive timeouts a connection should be abandoned.

9 Server Initialization and Startup

Initialization of the server is accomplished with the following function:

```
void tftpInit(struct ip_addr * ip, struct ip_addr * netmask, struct ip_addr * gateway, char
        * macAddr, XEmacIf_Config * emac);
```

This function initializes the TFTP server and assigns it the passed IP address, and MAC address. The `netmask` and `gateway` parameters are used for routing. See the code of the example application for how to fill the `ip_addr` structure.

The `emac` parameter should be passed the instance of the EMAC with which the TFTP server will be connected. See the code of the example application to see how to get an EMAC instance from the EMAC driver.

After the `tftpInit` function has been called, files should be registered as described in Section 6.

To start the server, the `runTftpServ` function should be called. It is passed the same IP address passed to `tftpInit`.

The server initialization and startup sequence can be easily followed within the code of the example application.

10 Customizable Parameters

Several parameters can be changed depending on what is required of the server. The server uses parameters from both `lwipopts.h` and `tftp.h`

10.1 Parameters within lwipopts.h

Below is a description of some of the parameters that are found in the `lwipopts.h` file mentioned in section 5.1.2. The LWIP library must be re-compiled for changes in these parameters to take effect. See section 5.1.2 for instructions on how to do so.

`MEM_SIZE` The size in bytes of the "RAM" memory available to LWIP. TFTP only uses "RAM" memory for error and ack packets and does not require much.

`MEMP_NUM_UDP_PCB` The number of pcb structures available for UDP connections. Each UDP connection requires one pcb. This parameters defines how many simultaneous connections the server is able to handle. `MEMP_NUM_UDP_PCB - 1` client connections can exits. One pcb is used for the listening connection.

`PBUF_POOL_SIZE` The number of blocks of memory available in the "pool" memory type. These are the blocks that are linked together shown in Figure 1.

`PBUF_POOL_BUFSIZE` The size in bytes of each of the pool memory blocks.

`LWIP_UDP` Enables the compilation of LWIP's UDP facilities. Must be defined as "1" for the TFTP server to compile.

10.2 Parameters within tftp.h

Below is a description of the definitions that are safe to be changed by the user. Other definitions within `tftp.h` should be left unchanged.

`TFTP_MAX_TIMEOUT_TICKS` Defines the duration of time that is considered a timeout. The unit of time used is the interval between `tftpTimeoutTicker()` function calls.

`TFTP_MAX_TIMEOUTS` Defines the number of consecutive timeouts the server will tolerate before a client's connection is dropped.

`MAX_FILENAME_LEN` The number of bytes allocated for file names for each file in the file registry.

`NUM_FILES` The number of files that can be stored in the file registry.

`LOWESTPORT` When establishing a connection with a client, a port within a range is chosen. This parameters determines the lower bound of the range of ports.

`NUMPORTS` The number of ports available for the server to pick from when establishing a client connection.

10.3 Debugging Parameters

The server contains one debugging parameter that can be used during compilation.

If `TFTP_DEBUG` is defined within the scope of `tftp.c`, the server will be compiled to print out extra debugging messages to the MicroBlaze's standard output.

`TFTP_DEBUG` can be defined within `tftp.h` or defined using command line options with the compiler as was the case in section 5.1.3.

11 Function Reference

```
void tftpInit(struct ip_addr * ip, struct ip_addr * netmask, struct ip_addr * gateway,
       char * macAddr, XEmacIf_Config * emac);
```

`ip` A pointer to a structure containing the local address of the TFTP server which the server will be listening on.

`netmask` A pointer to a structure containing the networking mask which defines which addresses need to be routed via a gateway.

`gateway` A pointer to a structure containing the IP address of the local gateway.

`macAddr` A 6 byte array representing the EMAC's MAC address.

`emac` A pointer to the XEmacIf_Config structure that defines the instance of the EMAC that will be used by the server. In general this is obtained from the EMAC driver

This function initializes the hardware and memory structures used by the server. It should be called before any files are registered and before `runTftpServ()` is called. The `IP4_ADDR` macro can be used to fill the IP address structures required for several of the arguments. See `examplemain.c` for example usage.

```
int registerFile(char * fileName, tftpCallBackFunc recvHandler, tftpCallBackFunc
       sendHandler);
```

`fileName` A null terminated string containing the file's name.

`recvHandler` A function pointer to the call-back function used when a data packet it sent from a client to the server for this file. This function should take the data as input. Can be `NULL` if the file is read-only.

`sendHandler` A function pointer to the call-back function used when a data packet of this file is to be sent from the server to a client. This function should fill the data buffer structure with output. Can be `NULL` if the file is write-only.

This function adds a file to the file registry. The two function pointers that are passed are called when data is being retrieved from or sent to the file. See section 7.1 for more information on the call-back scheme.
The function will return 1 on success and 0 if there is no more available space for files. The function should be called at least once before `runTftpServ()` is called. This function should be called after `tftpInit()` has been called.

```
int runTftpServ(struct ip_addr * myIP);
```

`myIP` A pointer to a structure containing the IP address that the server will be listening on. This should be the same structure that was passed to `tftpInit()`

Starts the TFTP server. The function does not return unless it has encountered an error. It will return 0 if it could not create the listening connection. It will return -1 if the listening port 69 is in use.

```
void tftpTimeoutTicker();
```

`void` This function takes no arguments

This function must be called on a regular interval to provide the server with its timing capabilities. Since `runTftpServ()` is a blocking function, this function in intended to be called from an interrupt service routing or another thread.