Phy100 v1.0

Phy100 is a spin-off from the GbPhy project, and is designed to operate with PHY devices limited to 10/100Mbps bandwidth which are typically present on lower cost evaluation boards such as the Digilent boards Nexys3 (SMSC LAN8710A) and Arty (Texas Instruments DP83848J). It provides a robust UDP-based Ethernet control link based on a fixed length 8-byte protocol whereby every 8-byte message always results in a corresponding 8-byte reply thus allowing the verification of, and correction for, eventual packet loss. For improved efficiency, up to 64 messages can be included in a single packet.

UDP addresses and ports

The control link is designed to allow multiple addresses selected by 3 bits of input, 'board_id', typically connected to DIP switches or header pins. With 'board_id' set to "000" the board responds to the IP address 192.168.1.10 with commands on port 10000. With 'board_id' set to "001" the board responds to the IP address 192.168.1.11 with commands on port 10001, and so on up to board_id "111". MAC addresses used are typical MAC addresses used by Xilinx, starting at 00-0A-35-00-01-02.

Hardware functionality

The project is structured as shown in Figure 1.

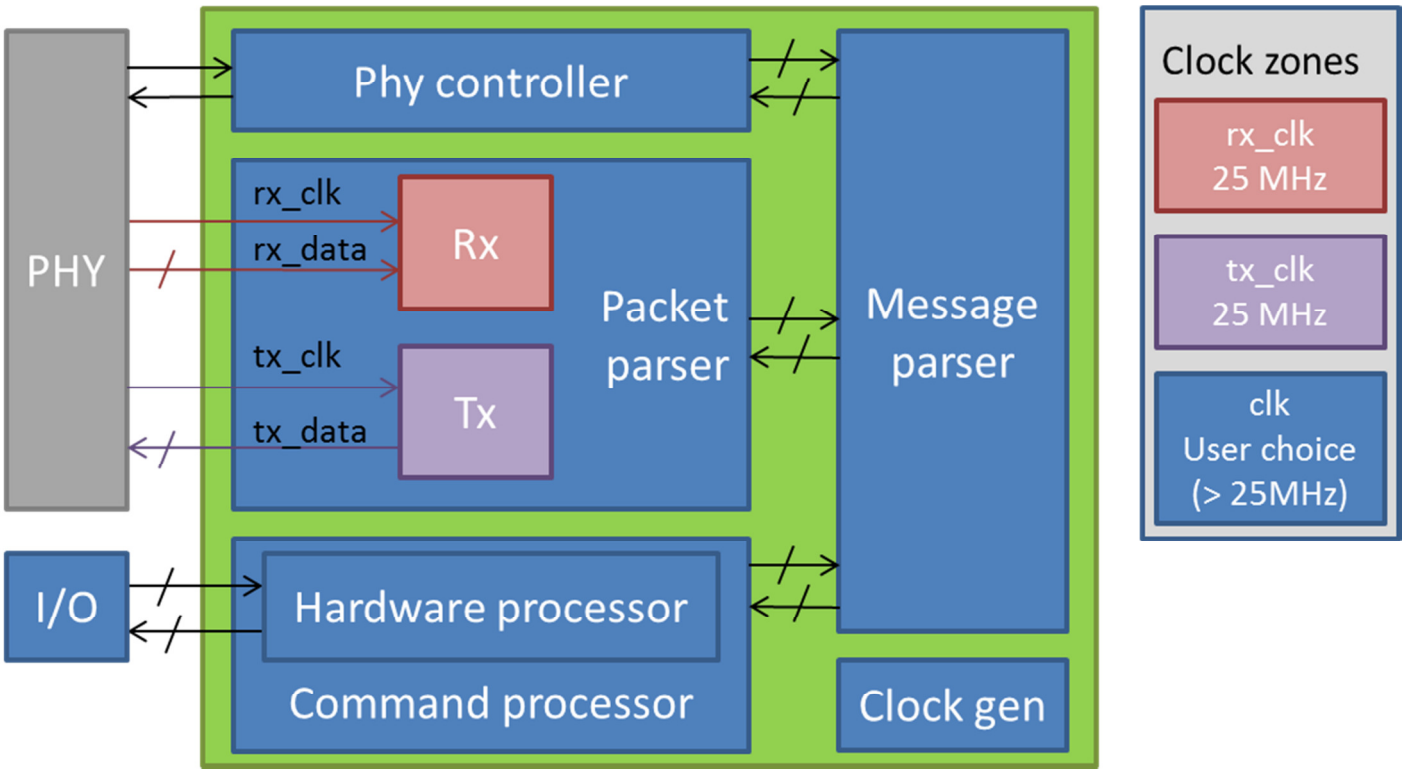

Figure 1: Example project schematic (in this case the 25MHz reference clock is supplied by a crystal oscillator on the board, alternatively the clock generator can be used to generate it)

The "Phy controller" block makes sure that the link is set to 100Mbps **Full** duplex at startup (for example, the Nexys 3 board only advertises 100Mbps Half duplex as default and there is no collision handling logic).

The "Packet parser" block interacts with the PHY device to receive and transmit Ethernet packets. It discards packets with incorrect IP or FCS checksums (the UDP checksum is ignored). It decodes and responds internally to ARP requests and ICMP ping. It passes the payload of correctly addressed UDP packets to the "Message parser" block. It does not respond to any other Ethernet protocols. When it

receives reply data from the “Message parser”, it encodes this data into a UDP packet and transmits it to the address contained in the corresponding command packet.

The “Message parser” block decodes the 8-byte protocol, maintains a copy of the last command packet received which can be read back, provides monitor functionality for Ethernet data quality, and allows programmatic interaction with the user logic. As previously noted, up to 64 commands can be concatenated in a single packet, all commands in a single packet will be executed followed by a single reply packet. The protocol was originally developed for projects using the Microblaze soft CPU to provide Ethernet functionality, with various software modules as well as hardware interaction. Essentially the first four bytes are software-related (and documented in the project file ‘network_commands’) and when set to the appropriate values will lead to the last four bytes being exchanged with hardware. As such the “Message parser” block connects to the user logic with the following ports:

```
command_available                   : out std_logic;
command_contents                    : out std_logic_vector(31 downto 0);
command_reply_available             : in  std_logic;
command_reply_contents              : in  std_logic_vector(31 downto 0);
```

Quite simply, ‘command_available’ is set to ‘1’ for one clock cycle when ‘command_contents’ contains a new command. The user logic must then act on this command and, once the operation is complete, it must provide its reply in ‘command_reply_contents’ and signal the availability of that reply by setting ‘command_reply_available’ to ‘1’ for one clock cycle. There must always be *exactly* one reply per command.

Command protocol

At this point the user is free to use command_contents and command_reply_contents as desired. However the example project goes on to provide further structure allowing for multiple hardware sub-system blocks. The 32-bit command from ‘message_parser’ is subdivided as

```
command_contents(31 downto 28) => sub-system target
command_contents(27 downto 20) => sub-system command id
command_contents(19 downto 0)  => sub-system command data
```

In the example project the only sub-system is the “hardware_controller”. Command codes are documented in the file ‘command_processor_codes’ and test programs are available in the Phy100 and Phy100Test LabVIEW projects.

Figure 2 shows an example of communication. In this case, four hardware commands are sent in a single packet and four corresponding replies are received in a single packet.

Figure 2: Command and reply strings

The commands used here are write-only commands and so, by construction, the reply is identical to the command. Note to users unfamiliar with LabVIEW that the strings are represented in “Hex display” format and are simply arrays of bytes (**not** ASCII, **no** delimiters).

The first command x”01010000102004FF” can be deconstructed as follows:

x”01”:	“CPU”
x”01” :	“Do FIFO command”
x”0000” :	‘packet_id’ (NB the user always sees zero but the communication VI’s handle this internally)
x”1” :	“Hardware” sub-system
x”02” :	“Hardware” command code
x”004FF” :	“Hardware” command data

Adding user logic

Figure 3 shows the project hierarchy.

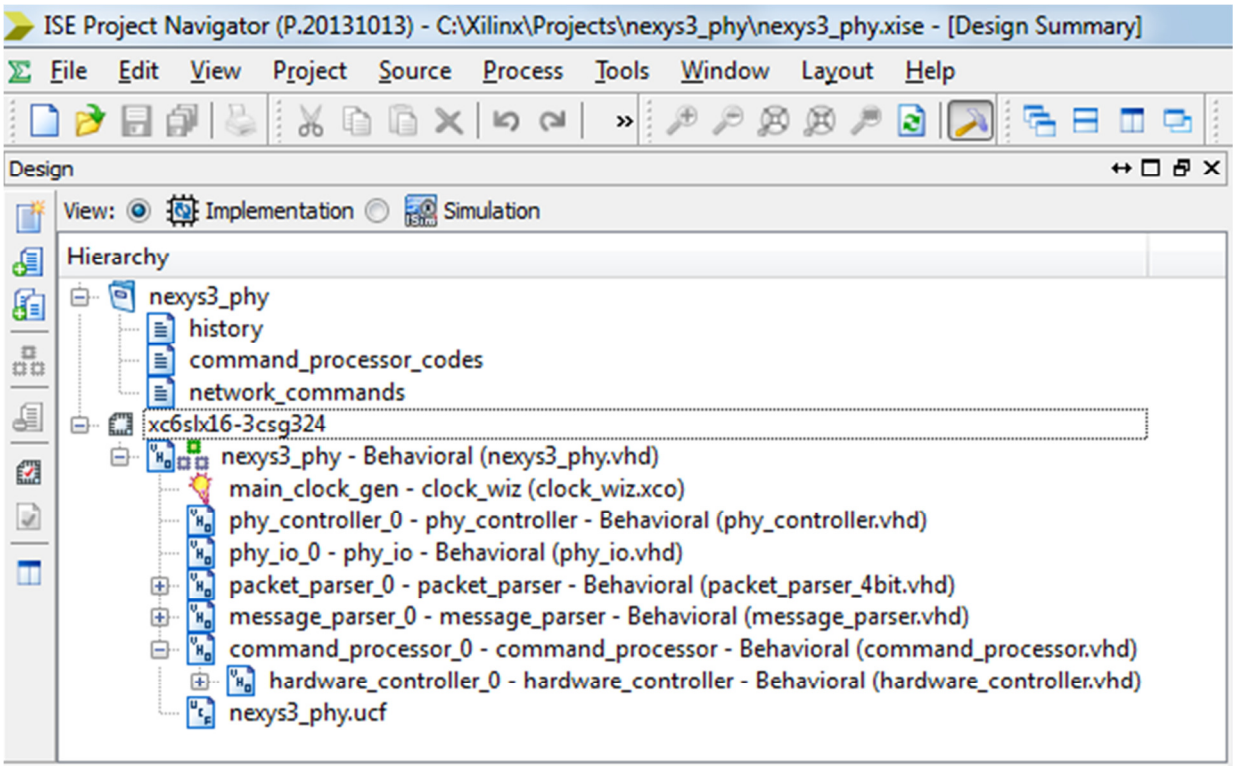

Figure 3: Project file hierarchy

The user can select the principle clock frequency by editing the clock_wiz IP core settings (it is recommended to use 100MHz or higher; the logic will certainly not work with the user clock below 25MHz). All custom user logic is added inside the “Command processor” and then connecting I/O up to the top level ports. The “Command processor” logic is declared in the file ‘command_processor.vhd’ for which there is an associated test bench ‘command_processor_tb.vhd’ which allows the user to simulate all custom logic without the need to generate or decode Ethernet packet streams. Any additional I/O must also be declared in the UCF file.

command_processor

The port map of 'command_processor.vhd' is as follows:

```
port (
       clk                                  : in  std_logic;
       command_available                    : in  std_logic;
       command_contents                     : in  std_logic_vector(31 downto 0);
       command_reply_available              : out std_logic;
       command_reply_contents               : out std_logic_vector(31 downto 0);
       GPIO_LEDs                            : out std_logic_vector(7 downto 0);
       GPIO_Switches                        : in  std_logic_vector(4 downto 0);
       GPIO_Buttons                         : in  std_logic_vector(4 downto 0)
);

clk                         : user clock
command_available           : set to '1' for one clock cycle by message_parser to indicate
                              that command_contents contains a new valid command
command_contents            : 32-bit command
command_reply_available     : set to '1' for one clock cycle by command_processor to
                              indicate that command_reply_contents contains a new valid reply
command_reply_contents      : 32-bit reply
```

Figure 4 shows the simulation of a hardware_processor read of the GPIO switches state

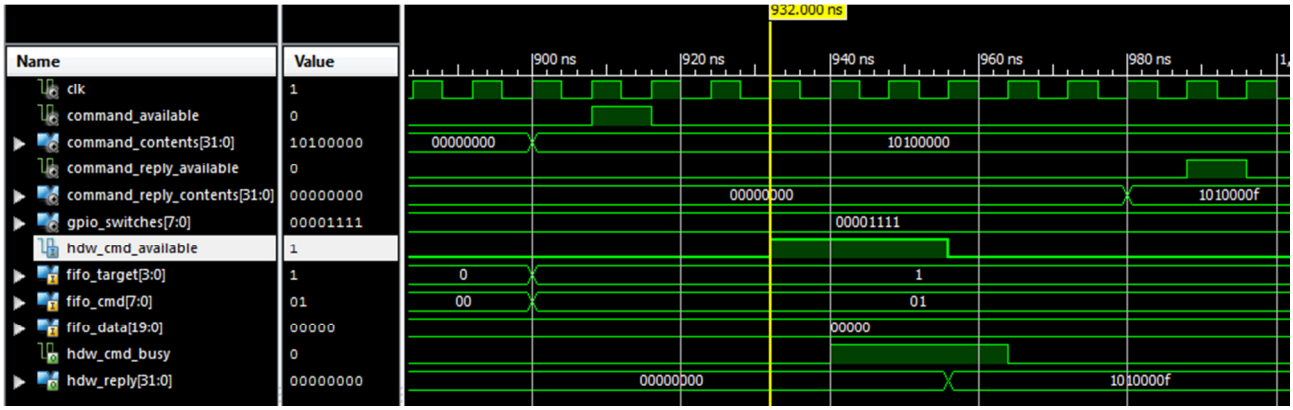

Figure 4: hardware_processor read

Adding user logic functionality

At the most basic level, user logic can be added to the 'hardware_controller.vhd' file. The starting point is the hardware command parser process called 'hdw_control_proc' and it should be obvious how to add extra functionality. Of course, other sub-systems can also be introduced starting from the 'do_cmd_proc' in the 'command_processor.vhd' file.

Network configuration

The firmware should, in theory, work on a shared LAN network, however there is absolutely no guarantee that it does. The only supported approach is to use a dedicated Ethernet port set as in figure 5:

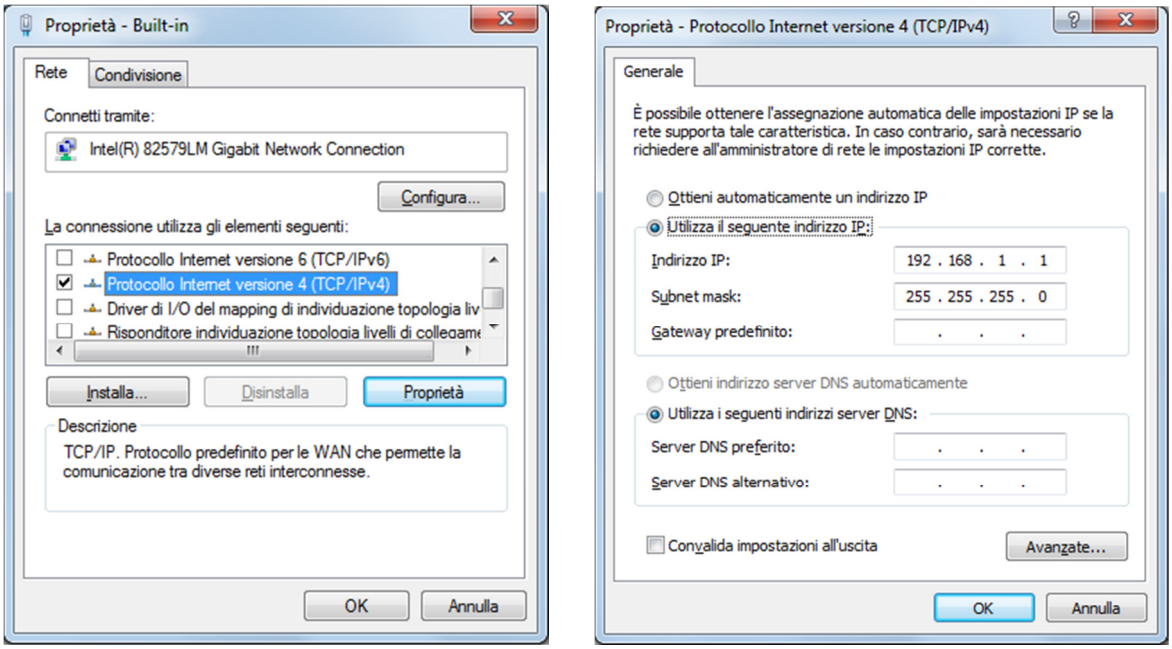

Figure 5: dedicated Ethernet NIC settings (Windows 7)

Note that all services except TCP/IPv4 are disabled and that there is no gateway defined. Multiple boards can be connected using a commercial switch.

LabVIEW driver and example projects

There is no absolutely no requirement to use LabVIEW, any programming language with access to UDP sockets can be used. However, only LabVIEW software support is provided here (divided into two projects).

Phy100 project

The Phy100 project (figure 6) contains all common “driver” functionality which will be used by all individual application projects.

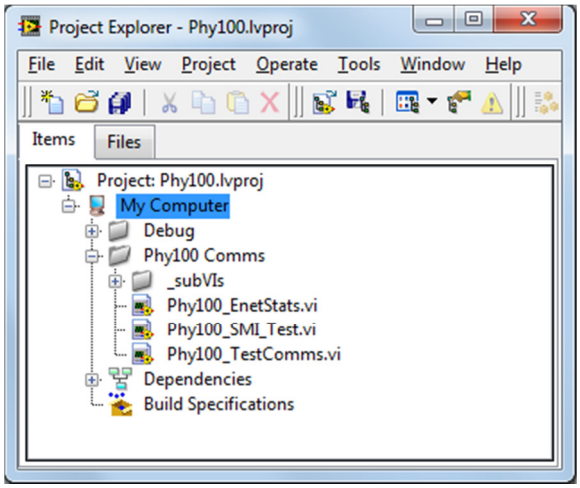

Figure 6: Phy100 LabVIEW project

The block diagram for ‘Phy100 _TestComms.vi’ (figure 7) shows the basic software approach

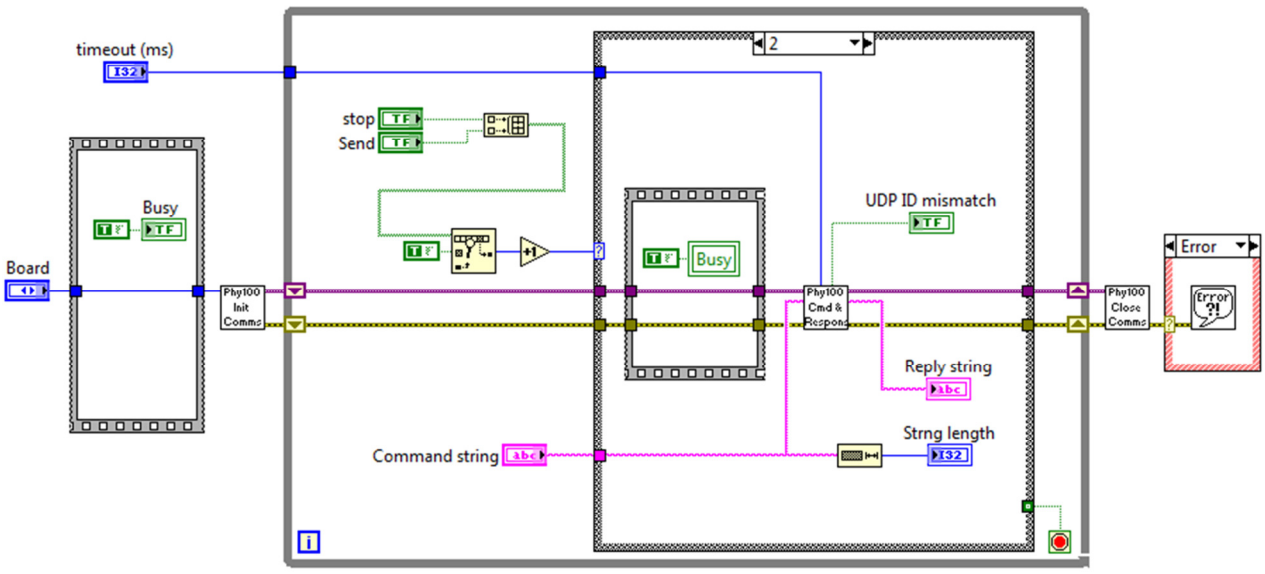

Figure 7: Phy100_TestComms.vi block diagram

To establish communications the user must call the VI ‘Phy100_InitComms.vi’ (figure 8) which opens UDP communication for the relevant port number.

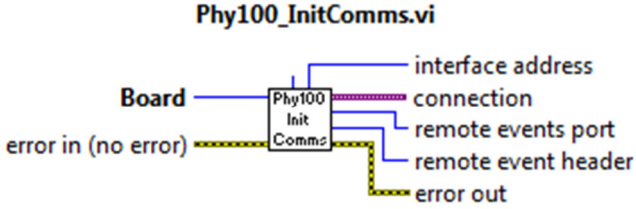

Figure 8: Phy100_InitComms.vi connector panel

The connection variant is only valid while the caller VI is running so the VI’s cannot be run “statically”, and since it also contains packet_id information it must be constantly passed ahead in loops via shift register.

The corresponding VI ‘Phy100_CloseComms.vi’ (figure 9) must be called before the caller VI finishes operation otherwise it may be necessary to close and reopen LabVIEW in order to regain control.

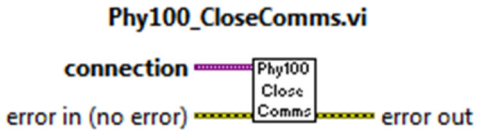

Figure 9: Phy100_CloseComms.vi connector panel

NB: the requirement to always call ‘Phy100_CloseComms.vi’ to close the connection implies that programs should avoid allowing the user to stop LabVIEW non-programmatically (by typing Ctrl-‘.’ or clicking on the toolbar “Abort” button). Experience suggests that the approach of hiding the toolbar when the program is running but leaving the menu bar visible with the Stop option available provides a good compromise between avoiding user error and allowing program abort in case of unexpected hanging of the program.

All command VI's call internally the VI 'Phy100_CmdAndResponse.vi' (figure 10)

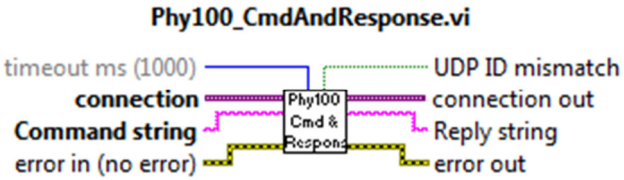

Figure 10: Phy100_CmdAndResponse.vi connector panel

'Phy100_CmdAndResponse.vi' handles communication with the FPGA including verification in case of reply timeout whether the preceding command was executed or not and therefore whether to resend or not. The user should treat this VI as a basic building block, simply providing input as LabVIEW string format with one or more eight-byte commands and extracting reply information from the output, as in figure 11.

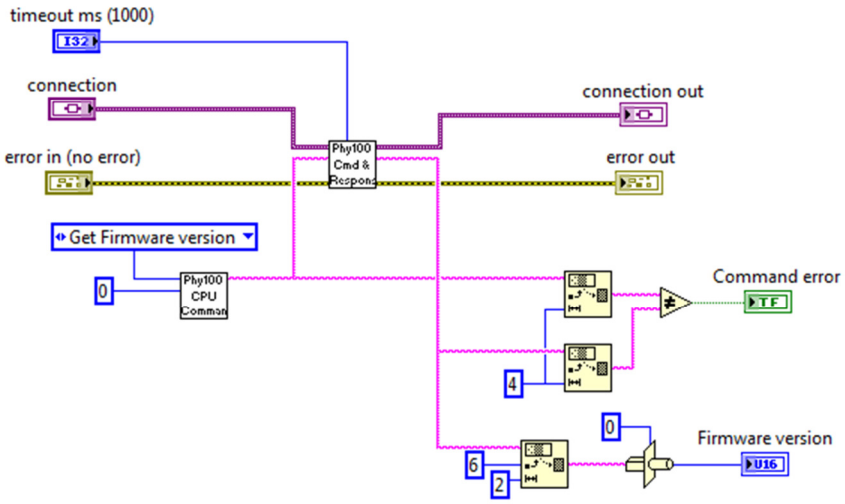

Figure 11: Usage example for Phy100_CmdAndResponse.vi

Phy100_EnetStats

Information regarding communication quality can be obtained using the 'Phy100_EnetStats.vi' (figure 12). Information on packet loss in the connection variant is saved to global variables by 'Phy100_CloseConnection.vi' so that 'Phy100_EnetStats.vi' can correctly report them so long as the other programs are stopped but not released from memory.

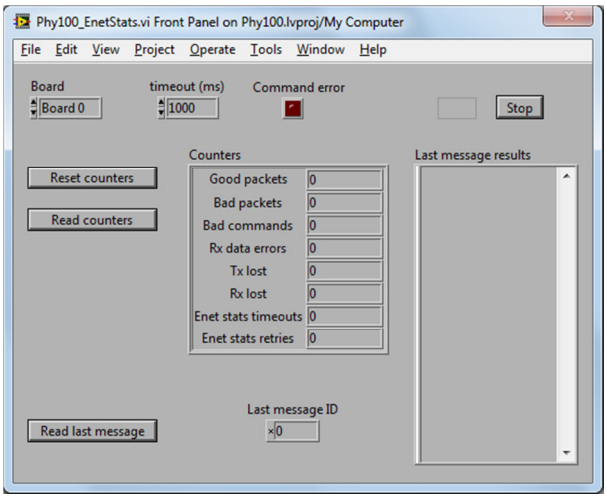

Figure 12: Phy100_EnetStats.vi front panel

The first four counters are hardware counters in the FPGA logic. “Good packets” indicates the total number of packets passing CRC checks and “Bad packets” indicates the total number of packets which failed CRC checks *since the board was programmed and regardless of whether those packets were addressed to the board or not*. These counters show whether the PHY/FPGA timing is good or not. “Bad commands” indicates malformed command strings (not multiples of eight bytes). “Rx data errors” indicates that the number of clock cycles for which the PHY output line “PhyRxer” was found to be ‘1’. “Tx lost” shows the number of packets sent but not received by the FPGA and “Rx lost” shows the number of packets transmitted by the FPGA but not received by the LabVIEW program. “Enet stats retries” shows the number of times the request for the last command executed had to be repeated due to packet loss.

NexysPhy project

The NexysPhy project (figure 13) contains the VI’s necessary to work with the user functionality implemented in the example project.

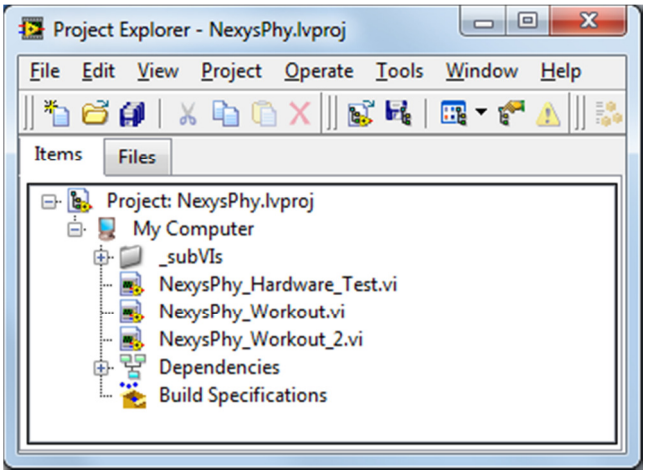

Figure 13: NexysPhy project

In the nexys3_phy example project the ‘hardware_controller’ logic provides functionality for reading the state of the GPIO switches and for setting the 7-segment displays (which by default display the firmware version) plus counter logic for packet loss debug. The VI ‘NexysPhy_Hardware_Test.vi’ (figure 14) implements these commands and provides the user with a starting point for software development.

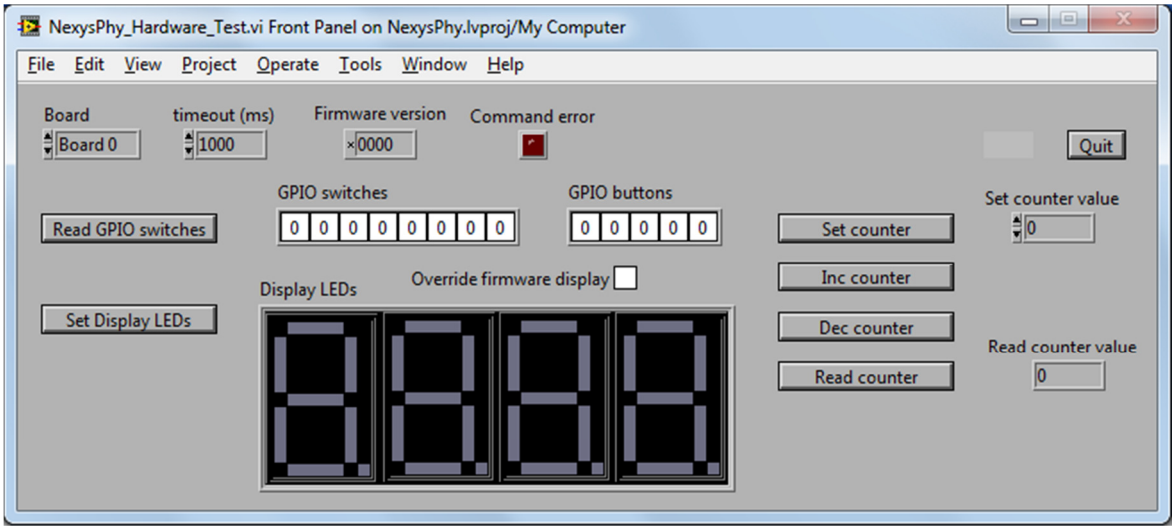

Figure 14 : NexysPhy_Hardware_Test.vi front panel

Figures 15 shows the block diagram of the VI 'NexysPhy_Hardware_ReadGPIOSwitches.vi'

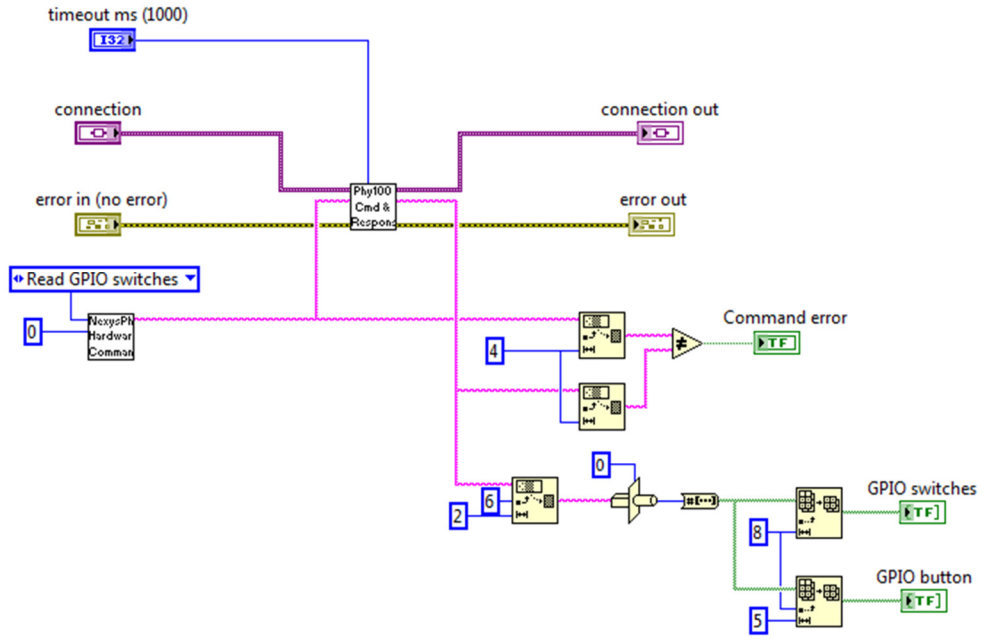

Figure 15: NexysPhy_Hardware_ReadGPIOSwitches.vi block diagram

Centre-left in figure 15 is the sub-VI 'NexysPhy_Hardware_Command.vi' used to generate the command string. While the use of a sub-VI is not fundamentally necessary (a simple string constant would be sufficient), it is used as a method for maintaining clean readable code that can be easily adjusted or updated. Figure 16 shows the block diagram of 'NexysPhy_Hardware_Command.vi':

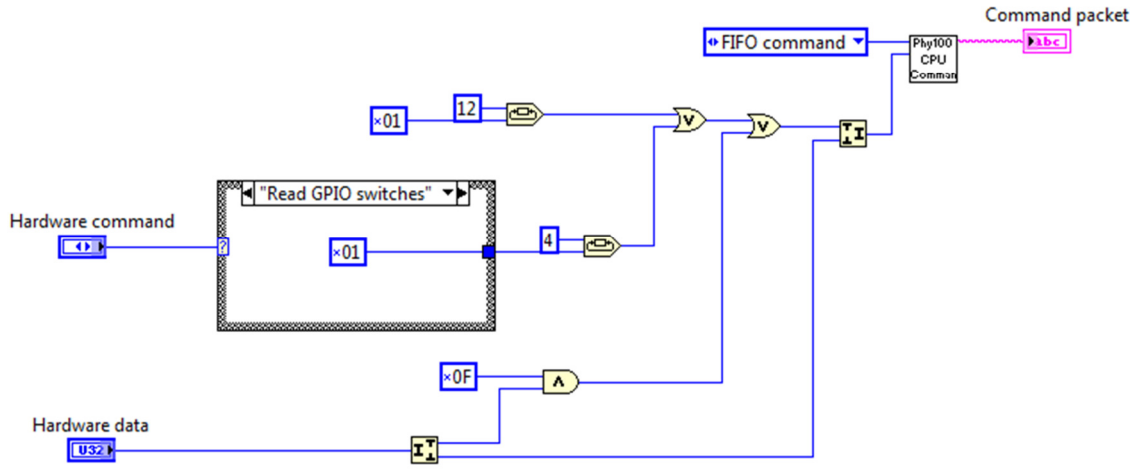

Figure 16: NexysPhy_Hardware_Command.vi block diagram

The "Hardware command" input is a strict type-def enum which lists all available commands, which indexes the case structure in order to obtain the appropriate command code. 'Phy100_CPU_Command.vi' (from the Phy100 project) then transforms the 32-bits of command into a complete eight-byte string configured to execute the "CPU" "FIFO command" operation which causes the 'message_parser' to interact with the user logic. Adding a new user hardware_processor command to the software therefore simply involves adding a new command to the 'NexysPhy_Hardware_Command.ctl' enum and a new command code to the case structure in 'NexysPhy_Hardware_Command.vi' (there is no default case so that any missing entries cause LabVIEW to signal an error).