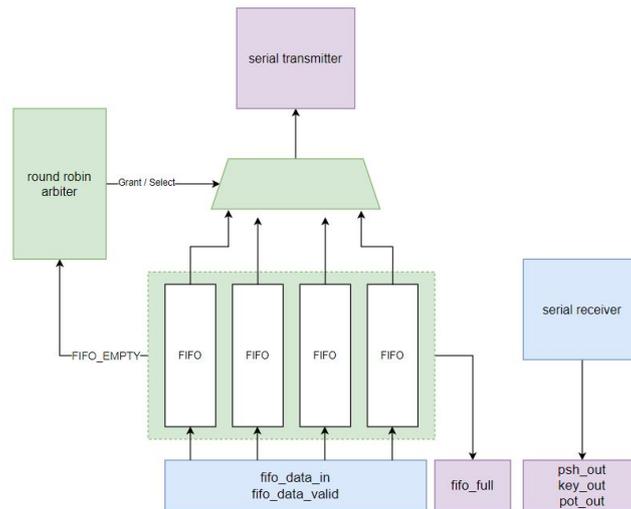


ECE 5760

Fall 2020

DE1-SoC Remote IO



Remote IO System Documentation

Author: Anthony Viego

Version: 1.0

Last Modified: December 7, 2020

Contents

1	Overview	2
1.1	What to Read	2
2	Adding Components to Qsys	2
3	Configuring the Top-Level	7
3.1	Instantiating the Qsys system	7
3.2	Pin Assignments and Project Files	9
4	Running and using the GUI	10
4.1	Functionality of the GUI	11
5	Quartus Interface - Details	11
5.1	Serial Protocol	13
5.2	System Infrastructure	14
5.2.1	Data Transmission	14
5.2.2	Data Receiving	15
5.2.3	Modifying the System	15
5.3	Modifying the GUI	16
6	References and Documents	16
6.1	Project Setup Guide	17
6.2	DE1-SoC User Manual	17
6.3	Quartus Handbook	17
A	Appendix	19
A.1	Serial Protocol Types	19
A.2	Serial Protocol Functions	19
A.3	FPGA System Overview	20
A.4	References	20
A.4.1	DE1-SoC Project Setup Guide	20
A.4.2	DE1-SoC User Manual	20
A.4.3	Quartus 15.1 Handbook	20
A.4.4	Avalon Interface Specification	20
A.4.5	Platform Designer (Qsys) Property Specification	21

1 Overview

This document is intended to provide you with all the information necessary to setup and use the remote IO system. This system consists of three main components, the virtual desktop running the GUI, the arduino acting as a serial pass-through, and the fpga. The steps necessary for setting up the system on each of these devices will be described in separate sections. A detailed description of how the system works is also provided in later sections.

In general, the remote-IO system is meant to provide you with access to IO which you would normally need to be in person to make use of. This includes seven-segment displays, LEDs, push-buttons, toggle switches, and potentiometers. A version of the GUI for this system can be seen in figure 1.

1.1 What to Read

This document consists of several sections, not all of which you will need to read depending on your situation. There are two primary audiences this document is intended for. The first is those who are using the PIO example project, and the second is for those looking to setup the system from scratch.

1. **Using example Project:** With the example project, everything should already be setup so there is no need to read through the sections discussing this. However, please do follow the first 4 points in section 2, to ensure that Qsys has found the IP path for the new files. Following this, refer to section 4 for an overview of how to run the GUI and what functionality it currently contains. Section 5 discusses the underlying infrastructure in place if you are looking to eventually add on to this system.
2. **Starting from scratch:** If you are starting from scratch with a brand new project, or simply adding the system to an existing project, then you will want to read sections (2-3) at the very least. Section 2 describes how to setup the necessary components in Qsys, and section 3 describes how to properly configure the top-level of quartus to handle the new Qsys modules. The remaining sections discuss the functionality of the GUI and the underlying infrastructure, so they are not strictly necessary for getting things setup and running.

2 Adding Components to Qsys

The first step in setting up this system will be adding two new components to QSys. Mainly the `io_avalon_interface`, and the RS232-UART. If you have not done so already, download the stand-alone zip file from the ECE 5760 website.

1. Within the zip file there should be three folders (GUI, arduino, FPGA). Place the FPGA folder into your existing quartus project folder.

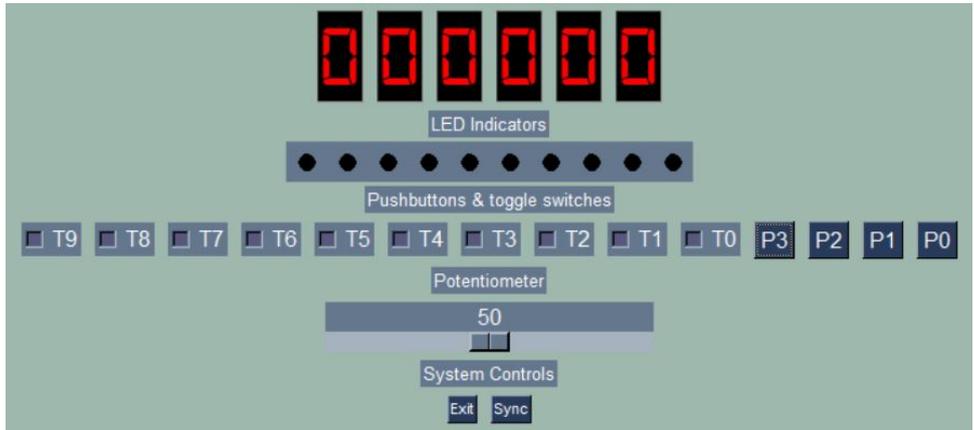
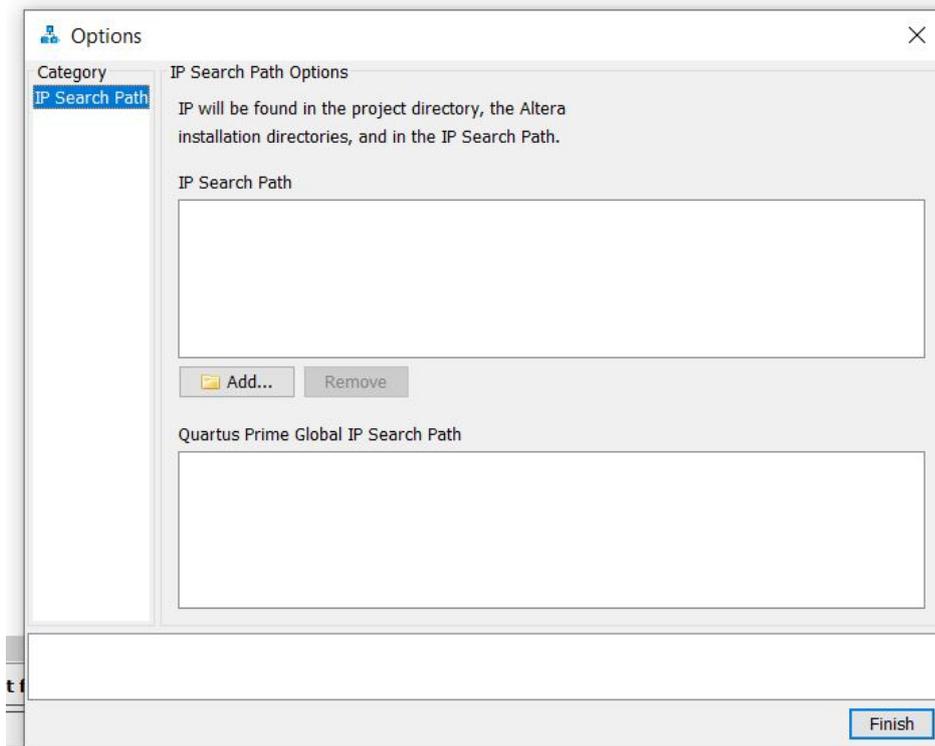


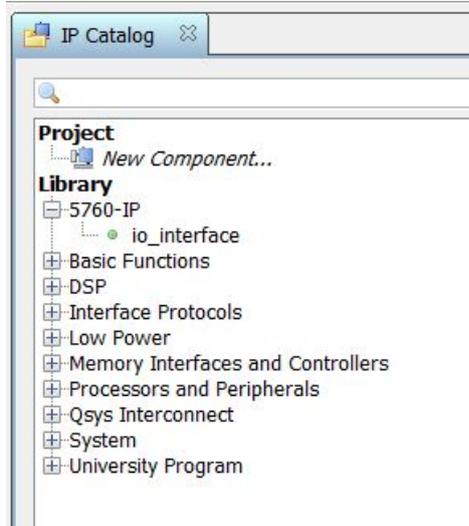
Figure 1: Remote IO GUI

2. Open your current project and go to qsys
3. In qsys go to tools– >options. This should open up a GUI that looks like the following:

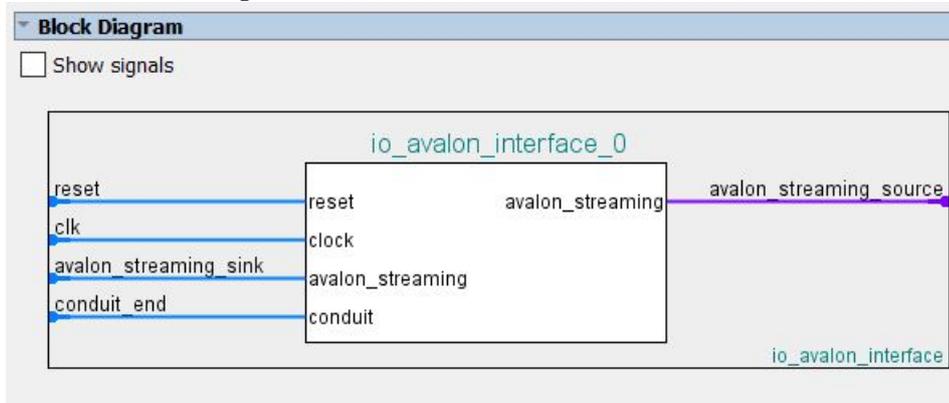


4. Click on add and select the FPGA folder that you added to your project folder earlier. Once done click finish. You should now see Qsys scan the folders and if

successful, the IP catalog on the left-hand side of Qsys will now show a 5760-IP category as shown in the image below:



5. The io_interface will be the first component we want to add to Qsys. Go ahead and click on the component and click add. This will open up a tab showing you the block symbol and the parameters. Ensure that the block symbol and parameters match those in the images below



The image shows a configuration window with three sections:

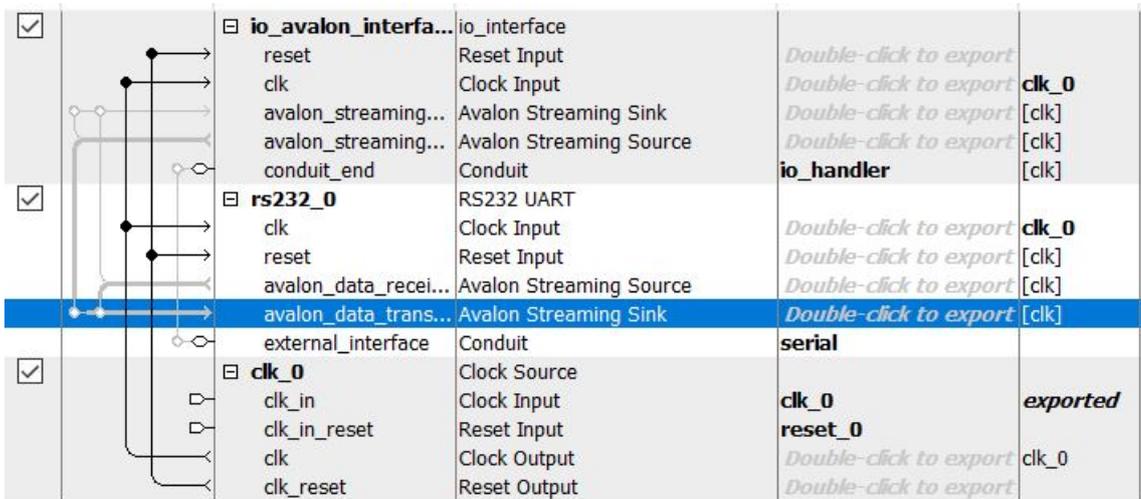
- FIFO Params:**
 - FIFO_depth: 8
 - FIFO_width: 64
 - GEN_FIFOS: 5
- LED Params:**
 - LED_timer: 0x0005f5e10
- Seven Seg Params:**
 - seven_seg_timer: 0x0017d7840

6. There will be a number of errors and warnings but we will fix these later. For now in the IP-Catalog go to University Program → Communications → RS232 UART. In the parameters that appear, change the avalon type to "streaming" and the Baud Rate to "38400". The parameters should now match the ones in the image below.

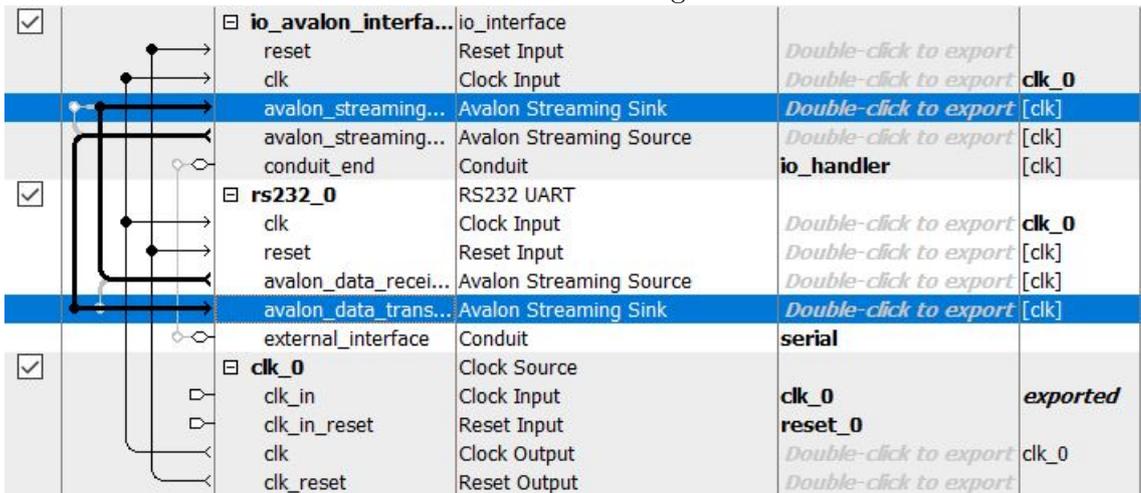
The image shows a configuration window with three sections:

- Interface Settings:**
 - Avalon Type: Streaming
 - Incoming clock rate: 50000000 Hz
- Baud Rate:**
 - Baud Rate (bps): 38400
- Data Format:**
 - Parity: None
 - Data Bits: 8
 - Stop Bits: 1

7. These are the only two new components that will be needed for setting up the system on the FPGA side. For the purposes of saving PLLs we are going to make use of the external CLOCK_50 the FPGA provides. For this step, insert a clock source from basic functions → clocks and plls. Make sure the clock frequency is set to 50MHz. Connect the clock output and reset output of this clock source to the clock and reset inputs of the io_interface and RS232 UART.
8. With the clock and resets setup we now want to export our conduits. A conduit in Qsys is an interface meant for data that isn't part of avalons predefined signal types (memory-mapped signals, val/rdy, etc). For the purpose of this setup, export the io_interface conduit as io_handler and the UART conduit as serial.
9. Your two blocks should now look like those shown in the image below. Note the exported names don't necessarily matter.



10. At this point we now need to configure the connections between the two blocks. The goal here is to have the streaming source of one block connect to the streaming sink of the other. Make sure that you do not configure the block to feed the data back into itself. The connections should look like the image below



11. With this complete. The last step is to simply generate the HDL using generate → generate HDL. You can then copy the instantiation template using generate → show instantiation templates. It should look similar to the image below if you are using a new project

```
computer_system u0 (  
  .serial_RXD          (<connected-to-serial_RXD>),  
  .serial_TXD          (<connected-to-serial_TXD>),  
  .io_handler_key_out  (<connected-to-io_handler_key_out>),  
  .io_handler_psh_out  (<connected-to-io_handler_psh_out>),  
  .io_handler_reset    (<connected-to-io_handler_reset>),  
  .io_handler_fifo_full (<connected-to-io_handler_fifo_full>),  
  .io_handler_fifo_data_valid (<connected-to-io_handler_fifo_data_valid>),  
  .io_handler_seven_seg_1 (<connected-to-io_handler_seven_seg_1>),  
  .io_handler_seven_seg_2 (<connected-to-io_handler_seven_seg_2>),  
  .io_handler_seven_seg_3 (<connected-to-io_handler_seven_seg_3>),  
  .io_handler_seven_seg_4 (<connected-to-io_handler_seven_seg_4>),  
  .io_handler_seven_seg_5 (<connected-to-io_handler_seven_seg_5>),  
  .io_handler_seven_seg_6 (<connected-to-io_handler_seven_seg_6>),  
  .io_handler_led_in   (<connected-to-io_handler_led_in>),  
  .io_handler_fifo_data_in (<connected-to-io_handler_fifo_data_in>),  
  .io_handler_pot_out   (<connected-to-io_handler_pot_out>),  
  .clk_0_clk            (<connected-to-clk_0_clk>),  
  .reset_0_reset_n     (<connected-to-reset_0_reset_n>)  
);
```

3 Configuring the Top-Level

This section focuses on instantiating the system defined in the previous step, and configuring quartus to define pins for our external connections.

3.1 Instantiating the Qsys system

If you have not done so already, copy the instantiation template from Qsys. This is an incredibly helpful tool that will ensure we do not miss any ports when instantiating our new system. Once copied, paste this template into your top-level design file. We will now begin to setup the connections necessary to get the design compiling.

1. To start with, assign a 1 bit 1 to both of the reset ports. As we won't have anyway of resetting the system manually, there is no need for us to connect an actual button to either reset signal.
2. Connect a 50MHz clock signal to clk_0_clk. If you are using a new project, define this clock signal as an input CLK_50. Make sure this is included in the port list as well.
3. With the clocks and resets taken care of, we now need to connect the serial ports. Define a 1-bit output port TX, and a 1-bit input port RX. NOTE: Don't forget to add these to the port list as well. Connect these two ports to the serial_RXD and serial_TXD connections in our instantiated module. Make sure to connect RX to RXD and TX to TXD. At this point your top-level should look something like the image below. Note: if you are using a new project, define a 10-bit output port LEDR as well. This will be useful for debugging.

```

module DE1_SoC_Computer (
    LEDR,
    CLK_50,
    RX,
    TX
);
input          CLK_50;
input          RX;
output        TX;
output [9:0] LEDR;

    computer_system u0 (
        .clk_0_clk          (CLK_50),
        .io_handler_key_out (0, //
        .io_handler_psh_out (0, //
        .io_handler_reset  (1'b1),
        .io_handler_fifo_full (0, //
        .io_handler_fifo_data_valid (0, //
        .io_handler_seven_seg_1 (0, //
        .io_handler_seven_seg_2 (0, //
        .io_handler_seven_seg_3 (0, //
        .io_handler_seven_seg_4 (0, //
        .io_handler_seven_seg_5 (0, //
        .io_handler_seven_seg_6 (0, //
        .io_handler_led_in (0,
        .io_handler_fifo_data_in (0, //
        .io_handler_pot_out (0, /
        .reset_0_reset_n (1'b1),
        .serial_RXD (RX),
        .serial_TXD (TX)
    );

```

4. For testing purposes we want some way of knowing that data is being passed from the GUI to the fpga, and vice versa. To accomplish this we will make use of the potentiometer and LEDs on the GUI. To start, create an 8-bit wire call POT and connect it to the pot_out connection on our module.
5. From here, assign the first eight bits of the LEDR port to our new pot connection, and then assign LEDR to the led_in connection on our new module. This will not only send the LED information back to the GUI, but also enable the LEDs on the fpga itself. Your top-level should now look like the following:

```

module DE1_SoC_Computer (
    LEDR,
    CLK_50,
    RX,
    TX
);
input      CLK_50;
input      RX;
output     TX;
output [9:0] LEDR;

wire [7:0] POT;

assign LEDR[7:0] = POT[7:0];

    computer_system u0 (
        .clk_0_clk          (CLK_50),
        .io_handler_key_out  (0),
        .io_handler_psh_out  (0),
        .io_handler_reset   (1'b1),
        .io_handler_fifo_full (0),
        .io_handler_fifo_data_valid (0, //
        .io_handler_seven_seg_1 (0, //
        .io_handler_seven_seg_2 (0, //
        .io_handler_seven_seg_3 (0, //
        .io_handler_seven_seg_4 (0, //
        .io_handler_seven_seg_5 (0, //
        .io_handler_seven_seg_6 (0, //
        .io_handler_led_in     (LEDR),
        .io_handler_fifo_data_in (0, //
        .io_handler_pot_out    (POT),
        .reset_0_reset_n       (1'b1),
        .serial_RXD             (RX),
        .serial_TXD             (TX)
    )

```

3.2 Pin Assignments and Project Files

1. If you were to try compiling this design now. You will notice that quartus will complain that there is no module "computer.system". This is because we need to add a file generated by Qsys to our project. To do this goto Project → Add/Remove Files. Once in this gui click the small .. icon and browse to the following file: computer.system/synthesis/computer_system.qip . Once selected make sure to click the add button near the top of the GUI and then click ok. You should now see the qip file listed in the project files. The qip file is a "quartus IP file" and specifies to quartus the file paths for all the verilog files associated with the IP it was generated for. This is useful as we do not need to manually add each file being used in qsys to the project.
2. From here, our next step is to run analysis and elaboration. We need to do this in-order to allow quartus to see what ports the top-level module has declared. This will be important for our next step. To run analysis and elaboration go to processing → start → analysis and elaboration.

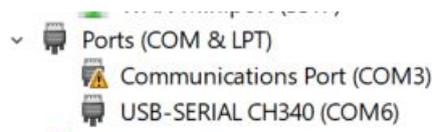
- Now that we have run analysis and elaboration, we want to make use of the pin planner to assign pins to our RX, TX, CLK_50, and LEDR ports. To save the time and trouble, there is a tcl script prepared already which will do these pin assignments. Navigate to tools → tcl scripts. Once here find the FPGA folder and select pin_assign.tcl With the script open click run and it will automatically assign the pins and the necessary voltage levels. If you now go to assignments → pin planner, you should see that every pin has a location and 3.3V I/O standard
NOTE: if you are not working from a new project but instead are using the existing DE1_SoC_Computer you should only need to go to assignments → pin planner and assign pins for RX and TX like in the following image. It should be noted that these pins may already be taken by GPIO_0[0] and [1]. Remove those assignments.

in	RX	Input	PIN Y17	4A	B4A N0	3.3-V LVTTTL
out	TX	Output	PIN AC18	4A	B4A N0	3.3-V LVTTTL

- At this point you can now compile your project and there should no longer be any errors. Once you have finished compiling, program the FPGA.

4 Running and using the GUI

At this point everything should be ready to go on the FPGA side. The only remaining steps are to check which COM port the arduino is connected to, and ensure the GUI program is attempting communication with this port. To begin check the COM port the arduino is on by going to device manager and scrolling down until you see the COM ports section. Here you should see a device listed as USB-SERIAL CH340 like in the image shown below. Make sure that the COM port that appears here matches the COM port specified on line 12 of gui.py and gui-fpga.py found in the GUI folder. It is recommended at this point to open the arduino code found in the arduino folder of the zip file, and flash the code to the board (remember to select the proper COM port in the arduino IDE).



Open a terminal of your choice and navigate to where you unzipped the GUI folder. You should now be able to run gui-fpga.py using the command (python gui-fpga.py). This should open a GUI similar to the one seen in Figure 1 on page 2. NOTE: if for some reason you get a python error talking about serial not having a write object, this means that the program failed to connect to the arduino. Once again, make sure that the COM port specified is the one for the arduino. Note, if you use the arduino IDE to open the "serial console", this will prevent the GUI from being able to use the COM port. You cannot have both the GUI and "serial console" open simultaneously. With the GUI now running you should be able to make use of the various buttons and switches available.

4.1 Functionality of the GUI

Here we will briefly talk about the functionality current available on the GUI. The image in Figure 1: shows the current GUI that opens when using `gui-fpga.py`.

1. **Seven-Segs:** The seven segment displays can be used to display information sent from the FPGA. Simply sending a binary encoded digit (0-15) to any of the `seven_seg` inputs will result in the data eventually being displayed on the corresponding seven segment display here on the GUI. These displays are actually just text elements which use a seven segment font, so adding additional displays for other uses should be simple. The font used also includes letters if one wanted to expand the functionality to allow for hex displays.
2. **LEDs:** The LEDs on the GUI are tied to the `led_in` port of the `qsys` module on the FPGA. When a bit on the input port is set high, the corresponding LED on the GUI will change from black (off) to red (on).
3. **Toggle Switches:** The toggle switches can be used just like the normal toggle switches on the `fpga`. Pressing a toggle switch will place a check-mark in the box and will update the `key_out` port to have a 1 for the corresponding switch. When the switch is pressed again, the bit will be set back to 0.
4. **Push Buttons:** the push buttons work similarly to the toggle switches. When the button is pressed and held down, the corresponding bit on the `psh_out` port is set to 1. When the button on the GUI is then released, the bit is set back to 0.
5. **Potentiometer:** The potentiometer sends an 8-bit value to the FPGA. This allows the user to act as if they have the output of a potentiometer hooked-up to an ADC. As such, by adjusting the slider, the port on the FPGA side will change accordingly. The example program created earlier in this document ties the LEDs on the GUI to this potentiometer so that we can see the values changing.
6. **Sync Button:** The sync button is a way of ensuring that the current state of the GUI is correctly represented on the FPGA. For example, if the GUI is closed or crashes when a toggle switch is currently high, this high state will remain on the FPGA, but the switch will appear to be low when the GUI is restarted. Pressing the sync button will send the status of the potentiometer and all toggle switches to the FPGA, so it can update its outputs accordingly.

5 Quartus Interface - Details

This section is meant to give an in-depth explanation of the system on the FPGA. It will mainly focus on explaining the currently implemented blocks and features, and describe the various ports currently defined in the top-level. This section is intended to allow the reader to gain an understanding of how to use the interface currently provided.

Port	Length	Description
clk	1-bit	Provides a clock signal for the system to use
reset	1-bit	reset signal for everything in io_interface
reset_n	1-bit	reset signal for clock generator
key_out	10-bits	contains the state of each toggle switch
psh_out	4-bits	contains the state of each push button
pot_out	8-bits	contains the number displayed on the potentiometer of the GUI
fifo_full	Variable	contains the full flag for each FIFO
fifo_data_valid	Variable	used to indicate data on the input of a FIFO is valid
fifo_data_in	Variable	used to insert data into each FIFO
seven_seg(1-6)	4-bits	used to insert data into a seven seg FIFO
led_in	10-bits	used to insert the state of each LED into the led FIFO
RXD	1-bit	UART serial input
TXD	1-bit	UART serial output

The table above gives a list and short description of the various ports on the io interface. Here, we will discuss a few of the ports needed to interact with the remote io system.

- seven_seg:** The seven_seg ports are meant for sending data to the GUI. Each port is used for sending data specific to that seven_seg display. (0 being the rightmost display on the GUI). It is important to note that GUI expects decimal numbers encoded as binary, NOT seven seg controls. Thus if you want to display a 2, simply put an 4'b0010 onto the seven_seg port. There is no need for a seven segment controller.

The rate at which the data on these ports is sampled is based on the seven seg timer parameter that appears in Qsys. As the clock we are using is a 50MHz clock, the default seven seg timer is set to 25,000,000. This means the data on the port is sampled and sent to the GUI every half second.
- led_in:** The led_in port is similar to the seven_seg port and is meant for sending the state of each LED to the GUI. Unlike the seven_seg ports however, there is only one LED port. The led_in port is 10 bits wide and as such, each bit is used to encode the status of one LED. If you want to say that LEDs 0 and 1 are high and the rest are low, simply send a 10'b0000000011.

Once again, the rate at which the data on this port is sampled is based on the led timer parameter that appears in Qsys. The default value is set so that the status of the LEDs is sent every eighth of a second.
- pot_out, psh_out, key_out:** These three ports are all fairly similar, and allow you to use the push buttons, toggle switches, and potentiometer as if you were in person.

pot_out is an 8-bit value which contains the binary equivalent of the value displayed on the GUI. key_out and psh_out represent the values of each push button and toggle switch on the GUI (high or low). This allows you to read these values just as easily as you would normally read the value of the buttons and switches.

4. **fifo_data_in, fifo_data_valid, fifo_full** the FIFO ports are the more complex part of the interface and are meant as a way for you to extend the functionality of the GUI. Lets say you wanted to add a new feature to the GUI (graph for example) and you want to send data to that graphing element, these FIFO ports are the way for you to do that.

The first thing to note about the FIFO ports is that they are variable in size. The size of each port is determined by the GEN_FIFO parameter from Qsys. This parameter allows you to easily change the number of general FIFOs without having to modify the underlying infrastructure. It should also be noted that you can change both the amount of data a FIFO holds (FIFO_DEPTH) and the width of each piece of data (FIFO_WIDTH). However, the system will not allow you to have a depth less than 1 or a width less than 32. **NOTE: DO NOT CHANGE THE FIFO WIDTH AT THIS TIME**

The width of the fifo_data_in port is $(64 \times GEN_FIFO)$. Every 64 bits will be the data in for a separate fifo. Thus if you want to send data into two fifos, one assignment would be from [63:0] and the second would be from [127:64]. The size of fifo_data_valid and fifo_full simply scale with GEN_FIFO. To push data into a FIFO you must set the corresponding fifo_data_valid bit on the clock cycle you would like to do so.

It is NOT recommended to leave the valid high, as you will quickly fill the FIFO and begin to lose data. The fifo_full bit corresponding to each FIFO can be used to determine when a fifo is full and is a helpful way of providing flow control. It should be noted however that sending data whenever the FIFO is not full may also be problematic since we are relying on the arduino to pass messages, and it may not be able to handle sending and receive data at extremely high rates. As such it is recommended to use a timer system to send messages every so often.

Hopefully from the explanation of the various ports you should now have an idea of what data is available on the fpga side and how you can send data to the GUI. But what about handling new data sent from the GUI to the fpga? The following section describes the serial protocol currently in place and how you can add to it to support new functionality.

5.1 Serial Protocol

The serial protocol currently implemented is fairly basic and mostly exists to provide a way to frame the data being sent back and forth. The diagram below shows how data is broken up in the existing protocol.

```
#packet layout
#-----
# start_char | type | payload_length | payload | end_char
#-----
```

1. **0x80 : start_char** The start_char is used to indicate to serial receivers that a message is beginning. This allows us to properly begin receiving a message without needing some sort of additional reliability mechanisms or handshakes
2. **type:** The type byte is used to indicate to a receiver what device/function a message is meant for. Receivers will then handle the payload accordingly based on the type of the message. A table of the currently supported message types can be seen in appendix A.1
3. **payload_length** The payload length byte is used to indicate to the serial transmitter how long of a message is to be sent. The transmitter will then send that many bytes followed by an end_char. This is also used by receivers to check whether an error in transmission occurred (the payload length field doesn't match with how many bytes were received before end_char was detected).
4. **payload:** The payload is dependent upon the type of message. Typically for a push button, the payload will contain which push button it is and what event is occurring (either a press or release). A table of events/functions can be seen in appendix A.2
5. **0x81 : end_char** The end_char is used to allow serial receivers to know that the current message has finished being received. Alongside the payload length this is one way of ensuring some sort of reliability since there is currently no re-transmission or check sums implemented.

5.2 System Infrastructure

This section will describe underlying infrastructure on the FPGA side. The goal here is to provide an understanding of how everything is working together for those who wish to expand upon its functionality or correct any errors.

5.2.1 Data Transmission

Appendix A.3 Shows an overview of the io_interface block. As can be seen the general infrastructure revolves around a series of FIFOs. These FIFOs take in data from fifo_data_in as well as the seven segment and LED data. The empty bits of the FIFOs are then combined to form the request vector of the round robin arbiter. This is a relatively fast arbiter which allows us to chose which FIFO to pull data from in a single cycle. The arbiter is also work conserving so there are no additional cycles when looping back around the last FIFO to the first one. The grant of this round-robin arbiter feeds into a

multiplexer which is then used to send data to the serial transmitter. It should be noted that data is only popped from a FIFO when the serial transmitter is ready to receive new data.

For those looking to expand on the existing data the FPGA sends, it is as simple as adding more general FIFOs via the parameters described in previous sections. The point of describing the transmission process is to present its weaknesses so that they can be avoided. Currently it takes roughly 0.2 milliseconds (based on 38400 baud) to send a byte of data (or roughly 1.6 milliseconds to send a full 64-bit packet). As such, data can only be taken from a FIFO roughly once every 1.6 milliseconds. If every FIFO in the system has data, it would take $(1.6 \times N)$ milliseconds to make one round trip of all the FIFOs. If the rate at which data is entering the FIFOs increases beyond this point, the FIFOs will begin to fill up and eventually data will be lost. To avoid this, it is recommended to make the rate at which you enter data into the FIFOs much slower, or make use of the `fifo_full` port.

5.2.2 Data Receiving

The serial receiver is where data being sent to the FPGA is handled. Data handling begins when the receiver reads in a `start_char`. Data is then continually stored in internal registers until an `end_char` is found. Once this occurs the serial receiver looks at the type of message, payload length and function and determines what to do with the data. In the case of toggle switches for example, it will change the state of `key_out` depending on which switch number was in the payload and if the function byte in the payload is a switch event. If the function byte was instead a sync event, a different action occurs. For those looking on how to modify the system to allow the FPGA to receive new events, this is where you will need to begin.

5.2.3 Modifying the System

Appendix A.4 shows the FSM for the serial receiver. The "PROCESS" state is where the serial receiver processes a received packet. In order to handle new data, you will need to add a new statement to this state in (`serial_receiver.sv`). Use the other statements as a guide on how to process your own data, as a note, the start character and end characters are not a part of the data being stored for processing. You will likely need to add additional ports to both (`serial_receiver.sv` and `io_avalon_interface.sv`) in order to get new data to the top-level module. When making these change it is important to note that you will need to modify the `io_interface` Qsys component in-order to get the changes to take effect. There are two ways you can accomplish this.

1. **Qsys Component Editor:** The component editor is a great way of editing Qsys components; however, I would not recommend using it for simply adding new ports to the conduit interface. This is due to the fact that when adding new ports, it tends to mess up the interfaces and reset parameter groupings. Additionally in Quartus 15.1 the component editor is bugged, and any syntax or verilog issues (even a missing semicolon) will result in a "null module error" with no other provided

information. If you would still like to use it, you can simply run analyze synthesis files in the files tab, and that should find your new ports.

2. **TCL file editing:** This is probably the simpler way to add a new port to a conduit. In the qsys folder which lives in the FPGA folder, you will find (io_avalon_interface_hw.tcl). This is the tcl script that tells Qsys which ports exist, what size they are, what parameters are there, etc. Lets say I wanted to add a new port called "NEW_IN" all I would need to do is add the following line to the bottom of the file.

```
add_interface_port conduit_end NEW_IN new_in Output 8
```

The above line tells Qsys that we want our internal port declaration of NEW_IN to become new_in when it gets exported via the conduit. Additionally, it tells Qsys that this port should be 8 bits in width. When you then generate the HDL in Qsys, it will take care of creating this port and you can add it to your top level instantiation.

5.3 Modifying the GUI

With an understanding of how data is sent and received on the FPGA side, lets now take a look at how we can modify the GUI to add new functionality. The GUI uses the python GUI library known as PySimpleGUI. It is recommended that one take a brief look at the examples and documentation provided by PySimpleGUI before attempting to modify the GUI. From reading the documentation you will discover that PySimpleGUI handles updates to the GUI through an event system. When a button is pressed for example, an event will be triggered, with the event name being equal to the "key" defined in the creation of the button. We use these events to then take an action. In the case of a button, we send a serial message corresponding to either the press or release of a button using the python serial library. If you would like to add addition functionality to the GUI, you will likely need to follow a similar process. Currently, the serial protocol byte definitions occur in protocol.py

In terms of handling new data being sent to the GUI, one should only need to add a new case to the handle_serial function. Previous functions should have already taken care of reading in a serial message and storing it in the serial_storage array. Once again the start and end bytes are not stored in this array, so keep this in mind when indexing.

6 References and Documents

This section is intended to go through various references/documents that were used in this project, and that may be helpful to anyone looking to create a similar project. A list of references can be seen in Appendix A.4.

6.1 Project Setup Guide

Appendix A.4.1 includes a link to a DE1-SoC project setup guide. This guide goes through what is required to setup a project from scratch in Quartus, including creating the empty project, doing pin assignments, configuring the top-level, and more. This guide was made for more recent versions of Quartus; however, most of what it describes is still relevant to Quartus 15.1

6.2 DE1-SoC User Manual

Appendix A.4.2 includes a link to a DE1-SoC user manual. This manual includes a lot of information on the usage of the board, from the various peripherals it has, to several example projects showing how to use the FPGA and HPS together. Chapters 2 and 3 were particularly useful for this system.

1. Chapter 2 shows the overall diagram of the board along with a diagram which shows which peripherals are connected to the FPGA, and which are connected to the HPS. This is useful as it allows one to see what devices the FPGA has access to without needing any sort of pin muxing in Qsys.
2. Chapter 3 goes into detail on each peripheral connected to both the FPGA and HPS. In cases where applicable, the manual also provides a circuit diagram of the peripheral, which is helpful in seeing what the default state may be (pull-ups to 3.3V in the case of the push buttons). Along with descriptions of each peripheral, the manual also provides the list of associated pins for use in the pin planner. This saves one the trouble of trying to figure out the pins arrangements themselves.

6.3 Quartus Handbook

Appendix A.4.3 includes a link to the intel Quartus II Handbook for version 15.1. This is a rather massive 1600 page guide which covers pretty much every detail about Quartus that you would want to know. Here we will mention a couple of sections that were particularly useful.

1. **Section 5: Creating Qsys Components - Page 265** This section discusses how to create a Qsys component in extreme detail, so what in here is useful to take a look at? For starters take a look at table 5-1. This table provides a description of each interface type available in this version of Qsys. If you are looking to create a new Qsys component, understanding the various interfaces and their use cases should help guide you in determining what you need to make your design function as intended. Here we will provide a brief description of each interface to help clear up any confusion:
 - (a) **Memory-Mapped:** The Avalon memory mapped interface is mainly meant to be used between a "host" and a series of "slaves". Each slave device will

consists of some sort of write-able or readable RAM/memory which is given an associated address range in Qsys. When the host wants to read from or write to these slave devices, it will specify an address on the Avalon bus, along with a serie of other signals. You can think of this interface as being similar to an I2C protocol, so if you are looking to setup communications between a master, and multiple other devices, this would be a way of doing that.

- (b) **Streaming (ST):** The Avalon Streaming interface is used for unidirectional transfer of data from one device to another. What this means is that data flows from the data source port of one device to the data sink port of another. While the interface is "unidirectional", a device can have both a data source and sink port, allowing for it to simultaneously read data in, and write data out. In terms of flow control, the interface is different from the memory mapped interface in that is uses a simple val/rdy protocol to allow device to know when data can be sent or received. If you only need to communicate from one device to another, this is the way to go.
- (c) **Conduit:** The conduit interface is different from the other two interface mentioned so far, in that it is more of a way to provide data sharing, rather than data communications. What is meant by this is that data put on a conduit is just like any other wire you might create in a file. The conduit interface has no val/rdy signals, or write enable signals, so it is up to the hardware on both sides to know when to sample data. As such, the conduit interface is meant to be used as a simple input/output port rather than a communications port.

The remainder of this section goes into great detail on the various types of Qsys components, and the ways to create them. For the purposes of simplification, there are really two main ways of creating and editing components in Qsys. Both of these methods were mentioned earlier in this document in section 5.2.3, but we will describe them in a little more detail here.

- (a) **Qsys Component Editor:** The component editor is a really useful tool for getting started on creating a new Qsys component. It is particularly useful in that it will allow you to generate a TCL script for a component simply by analyzing the HDL files you specify for your component. By then making use of the parameters and signals tabs, one can configure and create an entire Qsys module without the need for editing an TCL scripts. Additionally, page 278 of the Qsys handbook describes the various prefixes that can be used for naming ports in a HDL file. Using a prefix in the format of `¡interface type prefix¡_¡signal type¡` will automatically tell the component editor to assign that port to the interface of that type. This will make ones life much easier when using the component editor. Unfortunately however, the component editor does have its limitations. The parameter system that Qsys has is very cool, but its abilities are limited to assigning parameters to groups if you are using the

component editor. For anything outside of this, you will need to use a TCL script.

- (b) **TCL Scripts:** TCL scripts are used in Qsys to define the various interfaces, parameter configurations, display items, etc, that a module contains. As mentioned before, the component editor actually generates a TCL script when it is used, so it is recommended that one uses the component editor when first creating a component. However, once that initial TCL script has been generated, we can modify the TCL script to allow us to have more flexibility. As an example of what we can do, let say that you want to limit the range of a parameter, so that it can't take just any random value. We can accomplish this using the parameter property `ALLOWED_RANGES`, which is shown on handbook page 286. Furthermore, lets say you wanted to make a "optional port or interface", you can also accomplish this through modifications to the TCL script, like the ones towards the bottom of page 288 of the handbook. For those interested, appendix A.4.5 links to the section of the Qsys/Platform Designer documentation which discusses all of the available properties that can be used in scripting.

Outside of the ways to make a component and the types of interfaces, the section also goes into a lot of other details (including how to create composed modules / subsystems, dynamically generated parameters, and more), but those are topics that were not necessary for this system, and so will not be discussed here.

A Appendix

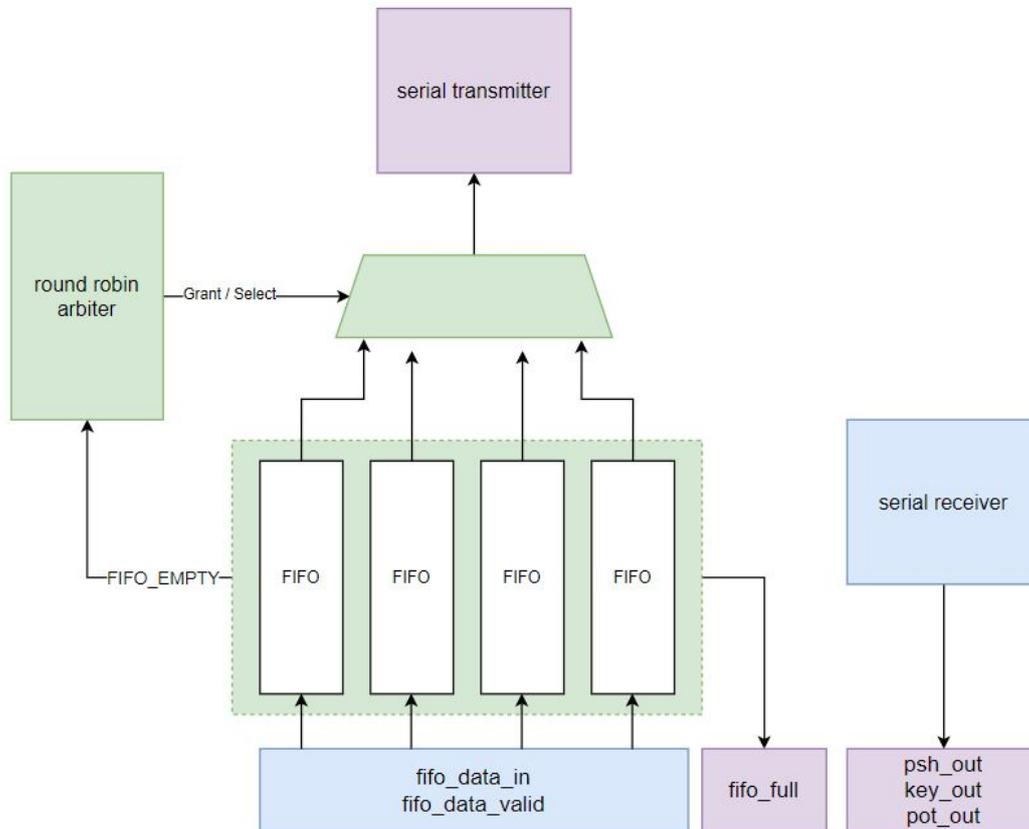
A.1 Serial Protocol Types

Char	Type
0x82	push button
0x83	toggle switch
0x84	key pad
0x85	seven segment
0x86	LED
0x87	potentiometer

A.2 Serial Protocol Functions

Char	Function
0x70	button press
0x71	button release
0x72	toggle (toggle switch)
0x73	sync

A.3 FPGA System Overview



A.4 References

A.4.1 DE1-SoC Project Setup Guide

PDF [Project Setup Guide](#)

A.4.2 DE1-SoC User Manual

PDF [DE1-SoC User Manual Revf](#)

A.4.3 Quartus 15.1 Handbook

PDF [Quartus Handbook](#)

A.4.4 Avalon Interface Specification

WEB [Avalon Specification](#)

A.4.5 Platform Designer (Qsys) Property Specification

WEB [Property Specification](#)