



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David



School of
Applied Computing

RS-232 SERIAL TO 5.0/3.3V VOLTAGE CONVERTER BETWEEN THE UART AND TCP PROTOCOLS

Final year independent project by Freya Vafadari

Abstract

My final year project is to design, build and test a TCP to serial data protocol converter device for RS – 232 to 0 3.3v/5v TTL. This is to enable a series of devices under test that take serial data as an input in RS 232 form and convert it into TCP data by changing the voltages to either 3.3v or 5v TTL or RS 232/12v. This is a requirement of a wider project in my organisation which needs to concurrently connect multiple serial devices, which don't have Ethernet connections, to an IP network so serial data can be generated and received across the network through a TCP connection.

Freya Jaleh Vafadari

Table of contents

Contents

Table of contents.....	1
List of figures	2
Introduction.....	3
Background information	4
TCP/IP transmission framework	4
UART protocol	5
RS -232 and 5.0/3.3V TTL	6
MAX 232 integrated circuit chip	7
Analogue multiplexer.....	7
Design methodology	8
Power supply choice	9
KiCAD	9
TXS0108EPW.....	10
ADG5209F	10
MAX232ECWE+T RS – 232 IC interface	11
DE 10 NANO FPGA with Cyclone V SoC	12
Intel Quartus Prime	13
Embedded Linux OS, device tree and root file system with Bookworm Debian	13
Serial to TCP bridge configuration and transmission in Python and HTML	13
Initial experimentation of an RS-232 to TTL voltage translation circuit using Proteus and CCS compiler.....	14
Hardware development of circuit and PCB design in KiCAD	16
Software development	21
Python Serial to TCP bridge and configuration software.....	21
Bookworm Debian Embedded Linux OS device tree and kernel.....	25
UART channel interface in VHDL.....	27
Testing methodology	32
Electrical design rule checking in KiCAD software	32
PCB testing.....	34
GPIO voltage software testing	37
Serial – TCP bridge testing using SOCAT virtual serial ports, NECAT and SCREEN tools	38
Serial port loop testing between Rx and Tx pins.....	39
System testing	40

System provisioning guide	42
Conclusion	43
Future improvements and development	44
References	44
Appendices	46
Component choices and information	46
Python bridge.py Serial to TCP bridge and configuration software	49
Embedded Linux device tree code	56
UART channel VHDL code	56
RS232 voltage translation with a MAX232 IC UART transmission in Proteus and CCS	58

List of figures

Figure 1 System architecture diagram.	4
Figure 2 TCP/IP framework.	5
Figure 3 UART protocol and timing diagram [9].	6
Figure 4 RS232 DB9 pin assignment [8].	6
Figure 5 RS – 232 threshold voltages. [8]	7
Figure 6 RS – 232 vs TTL serial comparison table. [8]	7
Figure 7 Analogue multiplexer symbol and switching function diagram. [3]	8
Figure 8 Project breakdown.	9
Figure 9 Power supply. [11]	9
Figure 10 TXS0108PW 8-bit bidirectional voltage-level shifter. [3]	10
Figure 11 ADG5209F analogue multiplexer. [2]	11
Figure 12 MAX 232 integrated voltage converter drivers and receivers' table. [1]	11
Figure 13 MAX232E integrated voltage converter pin out and application circuit. [1]	12
Figure 14 MAX232E pin out descriptions. [1]	12
Figure 15 DE10 NANO FPGA with Cyclone V SoC. [4]	13
Figure 16 Benefits and use cases of combining the implementation of pyserial and select libraries in Python.	14
Figure 17 RS232 voltage translation circuit with a MAX232 IC and PIC 16F877A in Proteus.	15
Figure 18 UART transmission code for the PIC 16F877A in CCS.	15
Figure 19 UART transmission test software flowchart.	16
Figure 20 KiCAD circuit diagram.	17
Figure 21 KiCAD PCB diagram.	19
Figure 22 Hardware photo.	20
Figure 23 Hardware photo with component and connector annotation.	21
Figure 24 Python functions and Flask routes descriptions.	Error! Bookmark not defined.
Figure 25 Python bridge.py software flowchart.	23
Figure 26 Python bridge.py thread and global dictionary structure and TCP protocol link.	24
Figure 27 UI example for serial port configuration.	24
Figure 28 Serial port and TCP server connection showing data flow directions of serial UART and TCP data.	25

Figure 29 Explanation of the device tree for the interfaces and GPIO (the code is shown in Figure 30).	26
Figure 30 Device tree source code for the FPGA UART and GPIO interface.	27
Figure 31 UART memory map summary.	27
Figure 32 VHDL signal declaration and combining signals.	28
Figure 33 Component instantiation mapping.	30
Figure 34 GPIO logic control of switches and MUXs.	31
Figure 35 UART schematic diagram from Intel Quartus Prime.	32
Figure 36 Initial use of the electrical design rules check feature in KiCAD with an incomplete circuit diagram.	33
Figure 37 Electrical design rule check feature in KiCAD after an initial run before any errors are corrected.	33
Figure 38 Proof of the circuit passing the electrical design rules check in KiCAD.	34
Figure 39 Rx and Tx signal flow in the PCB for 3.3v, 5.0v and 12.0v through the voltage converters and multiplexers switches.	Error! Bookmark not defined.
Figure 40 GPIO testing showing that a GPIO pin on the PCB can be driven HIGH and LOW.	37
Figure 41 GPIO testing showing that all 12 GPIO pins for a single channel are functional.	38
Figure 42 Testing using SOCAT virtual serial ports and NECAT tools.	38
Figure 43 TCP server running Serial-TCP bridge and showing transmit and receive of serial data.	39
Figure 44 TCP client running NECAT tool.	39
Figure 45 Serial interface shown using SCREEN tool.	39
Figure 46 Screen terminal in Linux showing serial test data.	40
Figure 47 DE10 nano FPGA and logic analyser during serial port testing.	40
Figure 48 Full system test of serial0_uart channel where the voltage = 5.0v.	41
Figure 49 Full system testing showing GPIO pin 0 driven HIGH to 5.0v.	41
Figure 50 Full system test of serial0_uart channel where the voltage = 12.0v.	42
Figure 51 Full system testing showing GPIO pin 0 driven HIGH to 12.0v.	42
Figure 52 Final year project Gantt chart April 2024 – May 2025. Initial plan in GREEN and actual time for some tasks in ORANGE.	43
Figure 53 Comparison of mains power supply. Option 1 [11] Option 2 [12]	46
Figure 54 Comparison of IEC embedded EMC filter. Option 1 [13] Option 2 [14]	47
Figure 55 Comparison of analogue multiplexer. Option 1 [2] Option 2 [15]	47
Figure 56 MAX232x IC voltage converter information. [1]	47
Figure 57 Mains power LED information. [17]	48
Figure 58 FPGA power LED information. [18]	48
Figure 59 10-way PCB mount. [19]	48
Figure 60 10-way cable mount. [20]	49
Figure 61 0.1uF ceramic capacitor. [16]	49

Introduction

My final year project is to design, build and test a TCP to serial data protocol converter device for RS – 232 to 0 3.3v/5v TTL. This is to enable a series of devices under test that take serial data as an input in RS 232 form and convert it into TCP data by changing the voltages to either 3.3v or 5v TTL or RS 232/12v. This is a requirement of a wider project in my organisation which needs to concurrently connect multiple serial devices, which don't have Ethernet connections, to an IP network so serial

data can be generated and received across the network through a TCP connection. An overall system architecture diagram is displayed below in Figure 1.

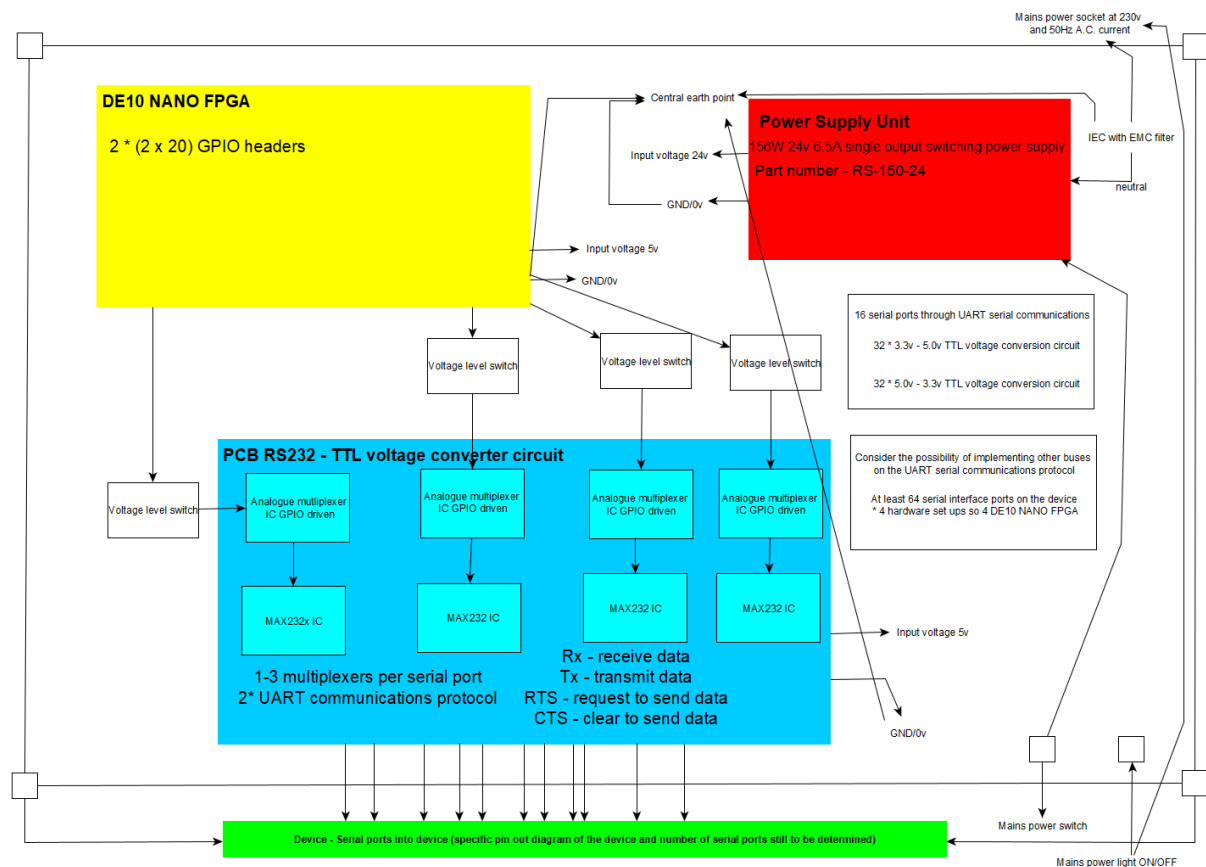


Figure 1 System architecture diagram.

Background information

TCP/IP transmission framework

The TCP/IP framework stands for the Transmission Control/Internet Protocol and is a set of standardised regulations that allow computer peripherals to communicate on an internet connected network. TCP and IP are two distinct computer network protocols where IP is the side of that obtains the network address of where the data packets are sent, and TCP is responsible for data transmission once the IP address is known. If data was transmitted in one piece, it's likely that it would be corrupted or fragmented and so all the data would have to be transferred again. However, TCP/IP separates each message into packets and the packets are reassembled once the data has been transmitted. The TCP/IP framework separates the different communication tasks into layer and each layer has a specific purpose. Data is transmitted through four individual layers before it is received at the other peripheral where the TCP/IP data is processed through the layers in reverse order to reassemble the data. The layer and protocol structure of TCP/IP are shown in Figure 2.

TCP/IP is a data link protocol that is implemented when transmitting data on the internet. The datalink layer handles the physical aspects of transmitting and receiving data for example through an Ethernet cable, wireless VPN network or a network device driver in the computer. The internet layer controls the packets movements inside the network. The transport layer forms a stable data connection between two peripherals and divides the data into discrete packets and sends and receives acknowledges of recipient of the packets. The application layer is a collection of

applications that need network connections like emails and instant messaging, a diagram of the TCP/IP framework is shown in Figure 2.

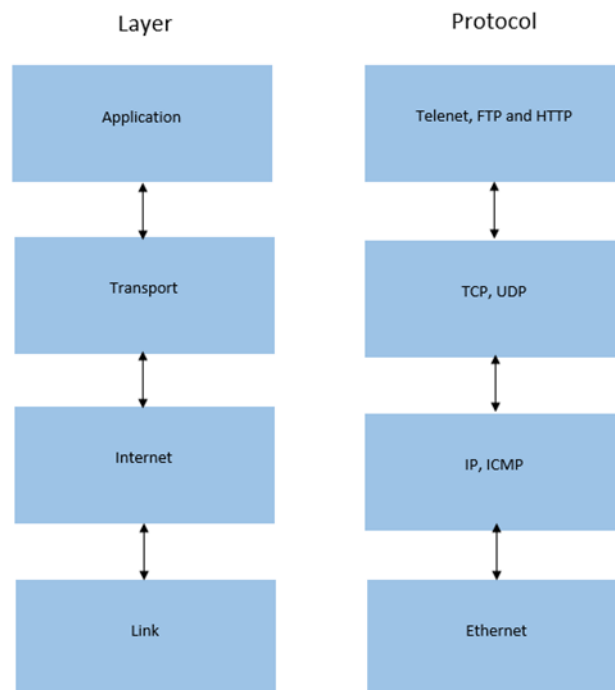


Figure 2 TCP/IP framework.

UART protocol

The UART or Universal Asynchronous Receiver Transmitter protocol is a standard for serial communication between devices and allows devices to send and receive one data bit at a time without the need for synchronisation of external clocks [9 & 10]. The two signals of each UART device are the transmitter (Tx) and receiver (Rx) and the function of these signals is to transmit and receive serial data for serial communication. The baud rate on both devices needs to be the same and is the rate of data transfer in the communication channel. If data is being transmitted from a serial port like it is in the case of my project the baud rate is the maximum number of bits per second being transmitted. UART transmits data in asynchronous form and so doesn't use a clock signal and transmit data as packets. A packet consists of a start bit, data frame, parity bit and stop bit. The UART data transmission line is 'high' when not transmitting and the data transfer starts when the state of the line changes from 'high' to 'low' for one clock cycle. When the receiving UART detects a state change, it starts reading the data bits at a frequency equal to the baud rate.

The data frame contains the data being transferred and is 5 – 8 bits long if a parity bit is present and up to nine bits if there's no parity bit. The parity bit is a check for the UART receiver to determine if any data has changed or been corrupted during transmission. After the receiver reads the UART frame, it counts the number of bits and checks if it's odd or even and if it's 0 there's an even parity and if it's 1 it's an odd parity. If the parity bit matches the data, then transmission is free from errors, however if it's 0 and the total is odd or 1 and the result is even then data has changed. At the end of the frame, there's a stop bit sent by the sending UART when the transmission line is driven from 'low' to 'high' for 1-2 bits. Figure 3 shows the bidirectional UART protocol with Rx/Tx lines and the timing diagram for UART showing that data is transferred one bit at a time.

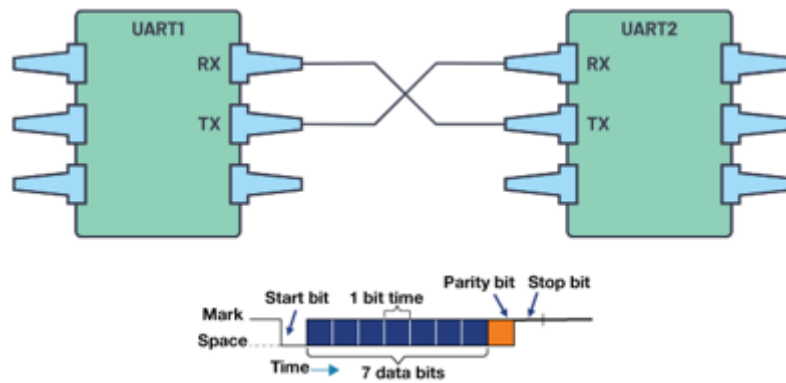


Figure 3 UART protocol and timing diagram [9].

RS -232 and 5.0/3.3V TTL

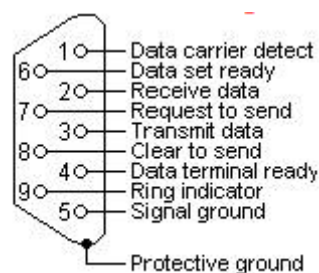


Figure 4 RS232 DB9 pin assignment [8].

RS – 232 is a standard protocol for serial communication utilised to connect computers and external peripherals [8]. The RS – 232 standard defines the electrical characteristics and timing of signals, behaviour of signals, physical size and pin out diagrams of peripheral connectors. Logic ‘1’ in the standard is in the voltage range -15 to -3v and logic ‘0’ is in the range +3 to +15v. Typically for RS – 232, logic ‘1’ is -12v and logic ‘0’ is +12v and all are defined with respect to a common GND. A voltage between -3V and +3V is in an undefined logic state. Figure 4 shows the pin assignment for a DB9 connector which is the most common RS232 connector.

In RS232, the data is transmitted serially in one direction over a single data line and at least three wires are needed for two-way communications (Rx, Tx and GND) in addition to control signals and so it uses start and stop bits to tell the receiver when to check for data. The individual signals on a serial port are unidirectional and if connecting between two devices, the output of one device must be connected to the input of the other device. The devices are separated into two types of data terminal equipment (DTE) like microcontrollers and data circuit terminating equipment (DCE) like an LCD display. In my project the DTE is the DE 10 NANO and the DCE is the connected device. In RS-232 the transmitter or the DTE sends a start bit to the receiver to inform it that data transmission is starting from the next bit and the 5-9 bits are data bits. If there is a parity bit, then a maximum of 8 bits can be transmitted otherwise 9 bits are transmitted and after this a stop bit is transmitted. TTL or transistor-transistor logic is a collection of digital logic circuits that utilise bipolar junction transistors to implement different logic functions. The fundamental block is the transistor which is a switch and can be formed of two different types of logic gates: NAND or NOR gates and are formed by connecting several transistors in a specific arrangement. Each logic gate in a TTL circuit will switch between two states: high (1) at 5v and low (0) at 0v representing binary data. TTL serial is a type of single ended serial communication using transistor voltage levels “high” for 1 and “low” for 0. One example of driver-receiver interface between TTL and long-range serial communication is the

MAX232 implementation which converts between TTL and RS-232. The RS-232 serial interface is in the range $\pm 3\text{V}$ - $\pm 15\text{V}$ with an undefined voltage state between -3V and $+3\text{V}$, the threshold voltage diagram for RS-232 is shown in Figure 5.

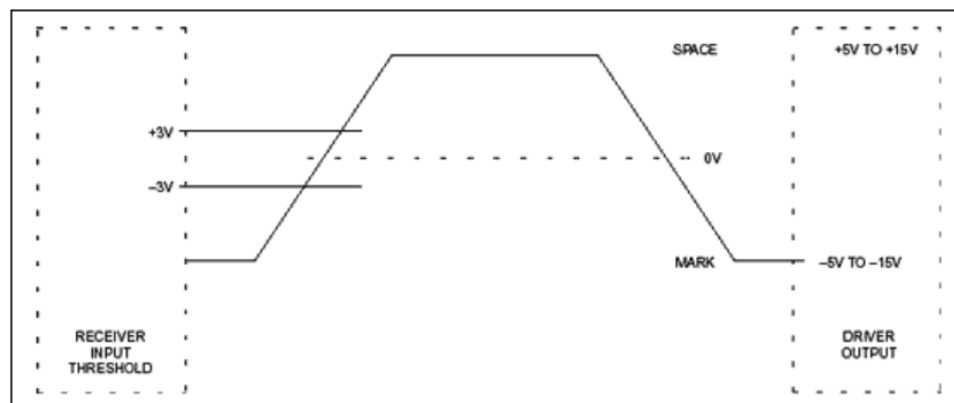


Figure 5 RS – 232 threshold voltages. [8]

Feature	RS 232	TTL serial
Overall voltage range	$\pm 3\text{V} - \pm 15\text{V}$	$0\text{V} - 5\text{V}/3.3\text{V}$
Logic 1 voltage	$-3\text{V} - -15\text{V}$	$2\text{V} - 5\text{V}$
Logic 0 voltage	$+3\text{V} - +15\text{V}$	$0\text{V} - 0.8\text{V}$
Undefined voltage state	Between -3V and $+3\text{V}$	Between 0.8V and 2V
Compatibility	Needs a voltage level converter to communicate with microcontroller	Direct communication with microcontroller
Use case	- Serial console interfaces like routers and switches - PLC	- Microcontrollers commonly have a 5V input power - Serial communication through UART - Voltage level shifters to change between 5V TTL and 3.3V CMOS

Figure 6 RS – 232 vs TTL serial comparison table. [8]

MAX 232 integrated circuit chip

I've chosen to use a MAX 232 integrated voltage converter for each channel because it's a common component to implement in circuits which require serial signals from a microcontroller at TTL logic levels to be converted to signals that are compatible with the RS – 232 serial communication standards. The MAX232 is a dual transmitter and receiver that is commonly implemented in circuits to convert Rx, Tx, CTS and RTS signals between RS – 232 and TTL voltages. In my project, a single channel comprises of the four separate signals Rx, Tx, CTS and RTS to form the UART protocol and each FPGA converts signals for a maximum of eight channels. The MAX 232 series IC chips have two receivers that change from RS-232 to TTL voltages and two drivers that translate from TTL to RS-232 voltages and, therefore only two out of four RS-232 signals can be converted in each direction. The first receiver/driver pair of the MAX232 used for Tx and Rx signals and the second pair for CTS and RTS signals.

Analogue multiplexer

An analogue multiplexer is a combinational logic device that allows several analogue input signals to be directed and transmitted to a single output. It functions as a switch that selects one of many inputs based on a control signal and send the selected input signal to the output. The control lines on a multiplexer are usually binary inputs which determine what input channel is connected to the

output. The multiplexer will switch between inputs based on the control signals and so the selected input is shown at the output. A multiplexer selects one of 'n' data inputs and forwards it to the output and the select one of several inputs so there is 'm' select lines such that $2^m = n$. Depending on the digital combination in binary that is applied at the select inputs then, one of the 'n' data inputs is selected and transmitted to the single output. Therefore, a multiplexer has 2^n data input lines, 'm' selects lines and one output line. The 4:1 multiplexer and the switching operation of the multiplexer from multiple inputs to a single selected output is represented in Figure 7.

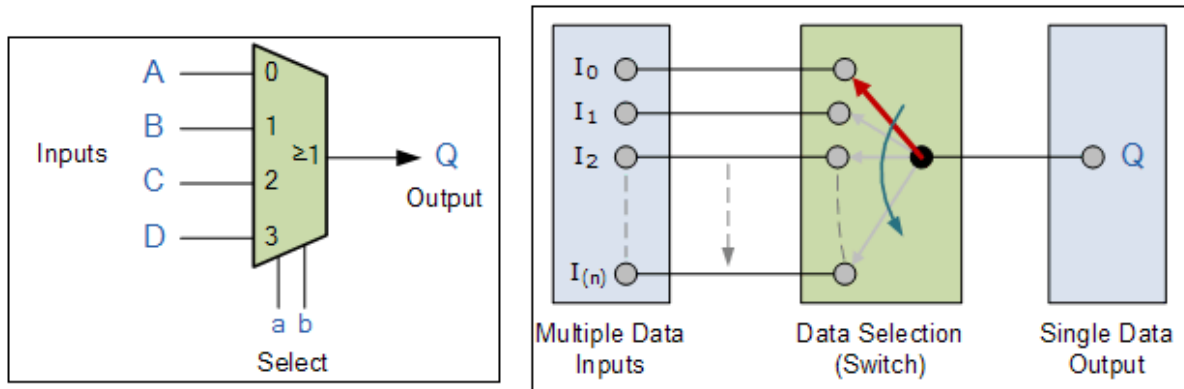


Figure 7 Analogue multiplexer symbol and switching function diagram. [3]

Design methodology

In the section below the main important design decisions for both software and hardware elements of the project are explained in detail and how those decisions were reached to develop a successful product which meets the requirement of the customer. The design decisions phase is an important stage in the project so the system can be effectively designed to meet the customers' requirements as they are currently but to allow the system to be easily modified in the future if the requirements were to slightly alter. For example, this is demonstrated in this project by the PCB be able to accommodate 4 individual UART serial channels using the Rx, Tx, RTS and CTS lines of the UART protocol however it would be possible to alter the signal function of the board so 8 individual UART serial channels are implemented using only the Rx and Tx lines if this was a requirement of the customer. Figure 8 shows the specific tasks in the project cycle that are required to successfully develop a functional and working system that meets the current requirements of the customer and allows for improvements and adaptations in the future.

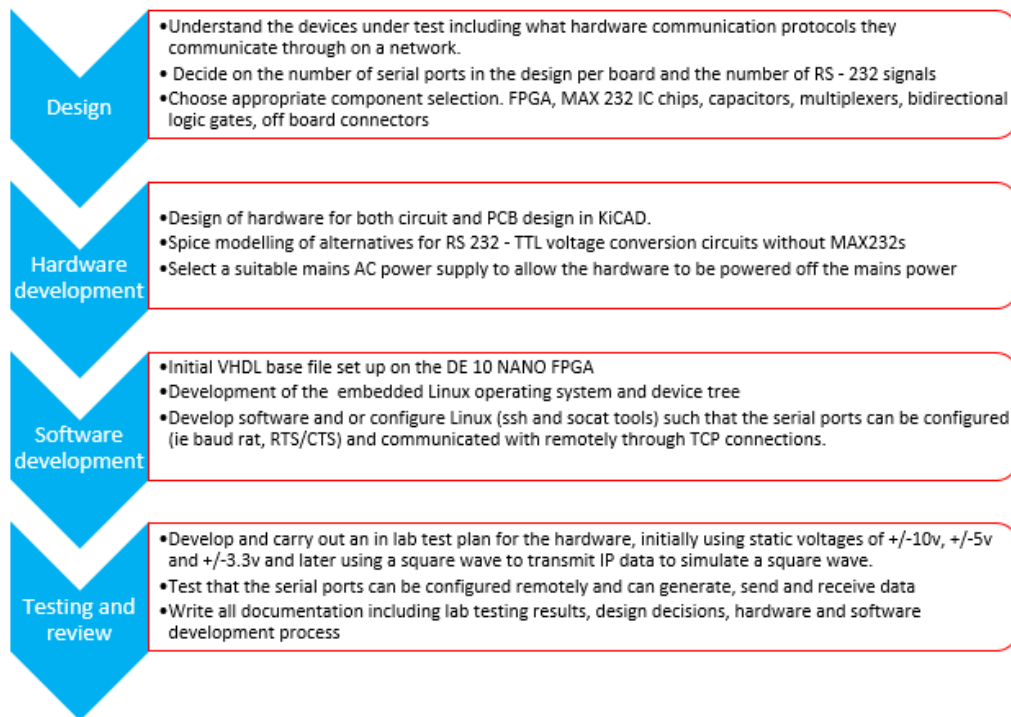


Figure 8 Project breakdown.

Power supply choice

I decided to select the Mean Well switching power supply from RS components which has the RS stock number – 644-7067 and Mfr part number – RD-50A [11] because it has the required output voltages for my system which are 5V DC @ 2A output current or 12V DC @ 6A output current and 12v is approximately the output voltage for RS-232 serial. I also decided to select this power supply specifically because its relatively efficient according to the product specification at 79% so doesn't waste a considerable quantity of heat when operating in the higher voltage ranges and can operate at a maximum temperature of +70°C.



Figure 9 Power supply. [11]

KiCAD

I've decided to design the 4-layer RS – 232 to TTL signal conversion board for between $\pm 12\text{v}$ and 0 – 3.3/5v TTL using the open source KiCAD design software. This software has been chosen for several

reasons because it's an open source requiring no license fees and is industry standard software that is extensively utilised to design custom PCBs for manufacture. KiCAD is an open-source software package that has a growing community of developers using the software around the world to design PCBs so there's several YouTube channels that were able to teach me the fundamental workings of the software and provide guidance especially in the later stages of designing when laying GND planes and cut out areas on the boards.

TXS0108EPW

The TXS0108EPWR from Texas Instruments is an 8-bit bidirectional voltage-level shifter designed to translate logic levels between different voltage domains and is suitable for high-speed data transfer and can support data rates up to 100 Mb/s [3]. The device operates across a wide supply voltage range from 1.2V to 5.5V. This device is an 8-bit non-inverting level translator which uses two separate configurable power-supply rails. The A port tracks the VCCA pin supply voltage and the VCCA pin supports any supply voltage between 1.4V and 3.6V. The B port tracks the VCCB pin supply voltage and the VCCB pin accepts any supply voltage between 1.65V and 5.5V. Two input supply pins allows for low voltage bidirectional translation between any of the 1.5V, 1.8V, 2.5V, 3.3V, and 5V voltage nodes. When the output-enable (OE) input is low, all outputs are placed in the high-impedance (Hi-Z) state. An application circuit of the TXS0108PW 8-bit bidirectional voltage-level shifter is in Figure 10.

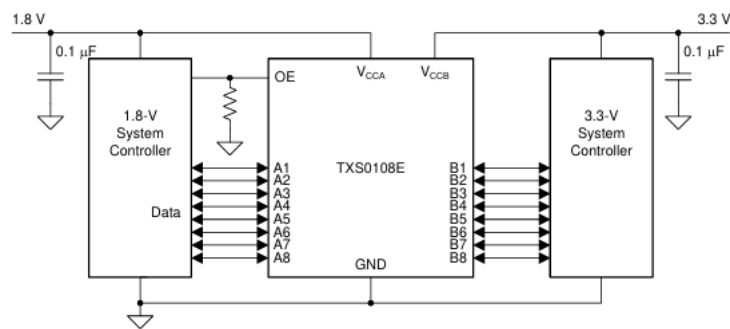


Figure 10 TXS0108PW 8-bit bidirectional voltage-level shifter. [3]

ADG5209F

I've selected the ADG5209F analogue multiplexer to switch one of four differential inputs to a common differential output [2]. An EN input on both devices enables or disables the device. The multiplexer is implemented in the circuit is to complete the switching operation between the 3.3V and 5V TTL voltages on the Tx and RTS signals in the UART serial protocol. I've decided to use the ADG5209f dual 4:1 analogue multiplexer in my design instead of the ADG5208F 8:1 dual analogue multiplexer because its only possible to have a maximum of four UART channels per PCB due to the constraints of the size of the FPGA header because each channel requires exactly 9 GPIO pins in addition to a 5.0V power line, 3.3V power line and two GND lines for each 40-pin header so there can't be any more than four channels per board. This means that 8 inputs aren't required to switch inputs to a common output. The ADG5209F analogue multiplexer can be operated from a dual power supply of $\pm 5 - \pm 22V$. The ADG5209F analogue multiplexer symbol is in Figure 11.

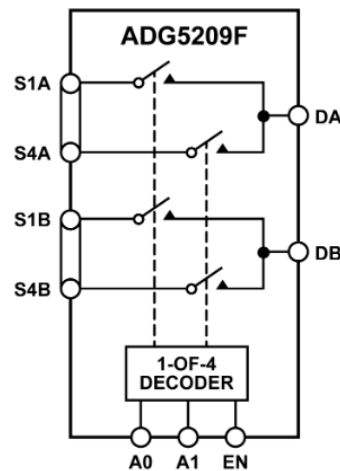


Figure 11 ADG5209F analogue multiplexer. [2]

MAX232ECWE+T RS – 232 IC interface

I've chosen to use a MAX 232 integrated voltage converter for each channel because it's a common component to implement in circuits which require serial signals from a microcontroller at TTL logic levels to be converted to signals that are compatible with the RS – 232 serial communication standards [1]. The MAX232 is a dual transmitter and receiver that is commonly implemented in circuits to convert Rx, Tx, CTS and RTS signals between RS – 232 and TTL voltages. In my project, a single channel comprises of the four separate signals Rx, Tx, CTS and RTS to form the UART protocol and each FPGA converts signals for a maximum of eight channels. The MAX 232 series IC chips have two receivers that change from RS-232 to TTL voltages and two drivers that translate from TTL to RS-232 voltages and, therefore only two out of four RS-232 signals can be converted in each direction. The first receiver/driver pair of the MAX232 used for Tx and Rx signals and the second pair for CTS and RTS signals. I chose to use the MAX232E component in my circuit because it had 2 RS-232 drivers and 2 RS-232 receivers which is the exact number of receivers and drivers that's required in my circuit to switch between Tx_3.3v to Tx_12v and RTS_3.3v to RTS_12v in the driver direction from TTL to RS232 and two receivers RX_12v to Rx_5v1 and CTS_12v to CTS_5v1 in the RS232 to TTL direction. Figure 12 shows the number of drivers and receivers in each integrated voltage converter in the MAX232xxx family. Figure 13 shows the pin out diagram and application circuit of the MAX232E integrated voltage converter. Figure 14 shows the RS232 pin descriptions from data and handshake signals from DTE or DCE end points.

PART	NO. OF RS-232 DRIVERS	NO. OF RS-232 RECEIVERS	RECEIVERS ACTIVE IN SHUTDOWN	NO. OF EXTERNAL CAPACITORS (µF)	LOW-POWER SHUTDOWN	TTL TRI-STATE
MAX202E	2	2	0	4 (0.1)	No	No
MAX203E	2	2	0	None	No	No
MAX205E	5	5	0	None	Yes	Yes
MAX206E	4	3	0	4 (0.1)	Yes	Yes
MAX207E	5	3	0	4 (0.1)	No	No
MAX208E	4	4	0	4 (0.1)	No	No
MAX211E	4	5	0	4 (0.1)	Yes	Yes
MAX213E	4	5	2	4 (0.1)	Yes	Yes
MAX232E	2	2	0	4 (1)	No	No
MAX241E	4	5	0	4 (1)	Yes	Yes

Figure 12 MAX 232 integrated voltage converter drivers and receivers' table. [1]

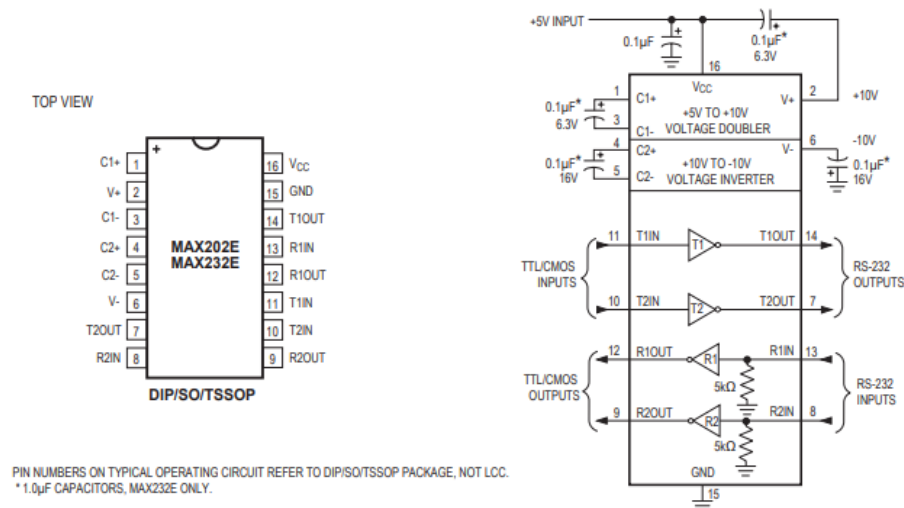


Figure 13 MAX232E integrated voltage converter pin out and application circuit. [1]

PIN	EIA/TIA-232E, V.28 SPECIFICATIONS	
1	Received Line Signal Detector (sometimes called Carrier Detect, DCD)	Handshake from DCE
2	Receive Data (RD)	Data from DCE
3	Transmit Data (TD)	Data from DTE
4	Data Terminal Ready	Handshake from DTE
5	Signal Ground	Reference point for Signals
6	Data Set Ready (DSR)	Handshake from DCE
7	Request to Send (RTS)	Handshake from DTE
8	Clear to Send (CTS)	Handshake from DCE
9	Ring Indicator	Handshake from DCE

Figure 14 MAX232E pin out descriptions. [1]

DE 10 NANO FPGA with Cyclone V SoC

I've chosen to use a DE 10 NANO to send and receive the serial signals in the system featuring an Intel Cyclone SoC FPGA which has an ARM processor [4]. I've chosen to use the DE 10 NANO because there are 80 GPIO pins separated into 2 * 40 pin headers which allows me to have eight IP to UART protocols lines for each FPGA because each channel requires exactly 9 GPIO pins in addition to a 5.0v power line, 3.3v power line and two GND lines for each 40-pin header. This means that for every FPGA, there is a maximum of eight IP to UART conversion channels between RS – 232 and TTL logic equally split between two 40 pin headers. The FPGA has 1 * 10 pin analogue input header which is also compatible with the Arduino analogue input lines. Figure 15 below shows the DE10 NANO FPGA with Cyclone V SoC and shows several main features of the FPGA which were important in the design choice. For example, it has 8 on board LEDs on the bottom right of the board which are beneficial features in the development of the UART channel interface in VHDL because it could used as a visual debugging tool when writing the signal interface in the Intel Quartus Prime software.

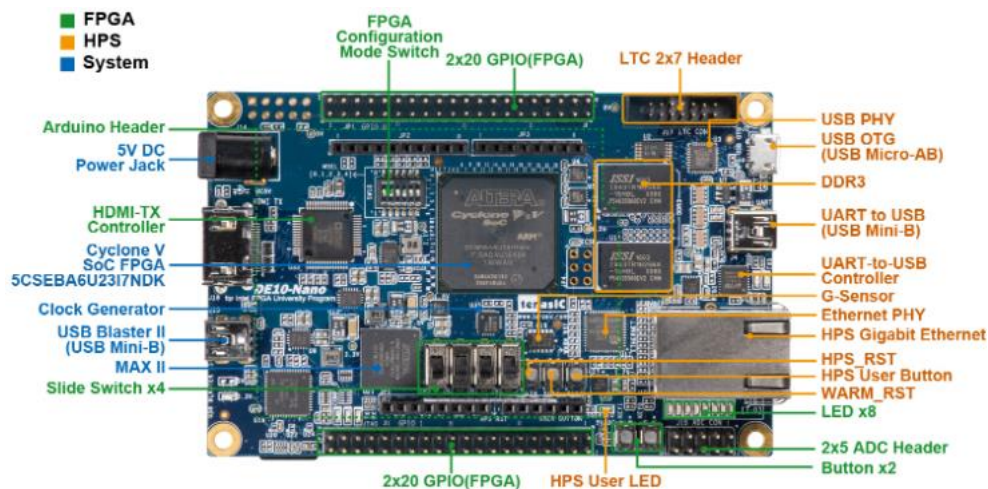


Figure 15 DE10 NANO FPGA with Cyclone V SoC. [4]

Intel Quartus Prime

I chose to write the VHDL signal interface for the UART channel using the Intel Quartus Prime FPGA software package because I am using the DE10 NANO FPGA with the Cyclone V SoC as explained above and as the Cyclone V is an Intel processor it was a sensible solution because the Lite version of the software requires no licenses and supports the Cyclone, MAX, Arria and Stratix SoC families. The automated interconnect generation in Intel Quartus Prime's platform designer is a beneficial feature because it streamlines the process of connecting various IP blocks within a system. It creates the interconnect logic needed for communication between components, ensuring compatibility and efficient data transfer. When a system utilises the Avalon UART IP block, Platform Designer integrates it by connecting the AXI interface signals to other components such as processors or memory and it automatically assigns unique memory addresses for each memory-mapped IP block. The AXI UART block is given a distinct base address and address range within the system's memory map, which ensures no overlap with other components.

Embedded Linux OS, device tree and root file system with Bookworm Debian

I developed the custom embedded Linux operating system and root file system with Bookworm Debian to build the OS, device tree and Linux kernel. I decided to use Debian 12 which is the Bookworm release because its stable well supported and has 59,000+ out of the box Linux downloadable packages that could be integrated if required. The Device Tree (DTB) provides the Linux kernel with a precise hardware description—including available peripherals (like UART, I2C, SPI, GPIO), interrupt mappings, clocks, power domains, and memory layout—which enables faster boot times, reduces kernel bloat, improves power efficiency, and simplifies debugging and maintenance by eliminating the need for generic board files and unnecessary hardware probing.

Serial to TCP bridge configuration and transmission in Python and HTML

I chose to develop the main Serial – TCP bridge configuration and transmission of serial and TCP data between the serial and TCP ports and TCP clients in Python because I've developed software extensively in Python before, so the language is familiar to me. I chose to develop the Serial-TCP bridge configuration and transmission software in Python because it enabled me to use the pyserial and select Python libraries. The pyserial library has several benefits that enable the development of serial communication software which include a simple interface easily opening, reading from, writing to, and configuring serial ports and is cross compatible across Windows and Linux platforms. The pyserial library allows for the development of safe multi-threads which has been implemented in my

code because I have a main thread which runs the web server, static file hosting and the `configure_ports` function in the main python script and the second thread runs the code which opens the serial and TCP ports and stores the file handles, GPIO voltage configuration and function to write the serial/TCP data to the appropriate serial/TCP port. The `select` library also has several benefits for example it allows multiple serial and TCP ports to be monitored at once and when one of the ports is available the data can be sent or receive only when the port is available. Figure 16 shows a table of the benefits and use cases of combining the implementation of `pyserial` and `select` libraries in Python.

Use case	Benefit
Data logger	Efficiently logs serial data only when there is data available
Interactive serial monitor	Non-blocking user input and serial read
Multi serial device communication	Monitor and communicate with several devices simultaneously
Responsive GUIs or web servers	Keeps GUI/server responsive while handling serial data in the background

Figure 16 Benefits and use cases of combining the implementation of `pyserial` and `select` libraries in Python.

I decided to write the front-end Python and HTML for the web server UI using the Python library `Flask` because `Flask` is one of the most popular micro web frameworks in Python and is simple to understand which was a benefit in making the decision because I've previously not had a significant amount of experience in developing front end software particularly UIs. `Flask` was also a suitable choice because it enabled a web server to be developed and hosted for the UI as opposed to a GUI which wasn't a possible option for this project as the GUI would've been unable to run on the CPU on the FPGA due to there being no GPU present.

Initial experimentation of an RS-232 to TTL voltage translation circuit using Proteus and CCS compiler

Some initial experimentation of interfacing the MAX232 integrated voltage converter with a microcontroller in this case with a PIC 16F877A occurred to understand how the MAX232 translates voltages from RS232. Figure 17 shows RS232 voltage translation circuit with a MAX232 IC and PIC 16F877A in Proteus and successful UART transmission from the PIC through the MAX232 IC as 'Transmit Hello World' is printed on the virtual terminal screen where the Rx and Tx lines of the virtual terminal are connected to the T1IN and R1OUT UART connections on the MAX232. The simulation was an initial experiment to understand more about how a MAX232 IC works and transmission of RS232 serial data. A PIC 16F877A was chosen to be the microcontroller in this circuit as opposed to the DE10 nano FPGA because its simpler to interface a single UART channel with a PIC and because Proteus doesn't support the simulation of FPGAs and can't generate a VHDL bitstream. Figure 18 shows the embedded C code in CCS for the RS232 voltage translation using a PIC 16F877A and a crystal circuit that has a frequency of 20MHz. The UART protocol has been set up to have a baud rate of 9600bps, transmission of 8 data bits and 1 stop bit. Figure 19 shows the flowchart for the UART software written in the CCS compiler.

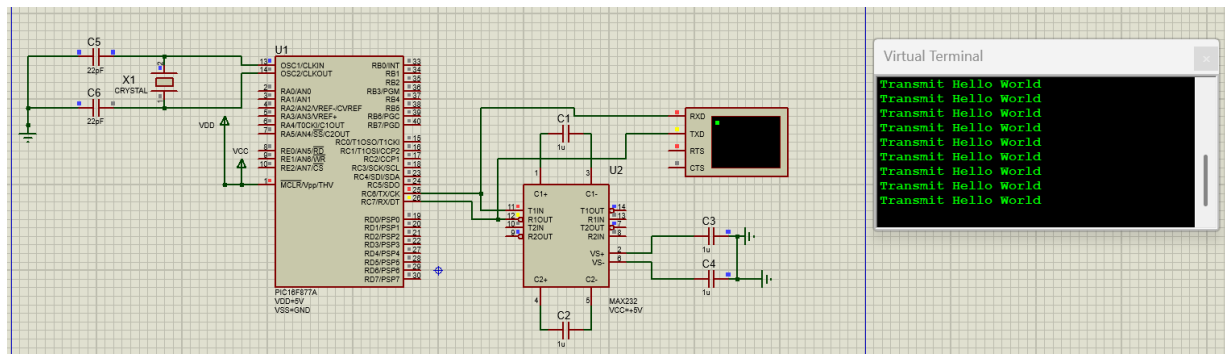


Figure 17 RS232 voltage translation circuit with a MAX232 IC and PIC 16F877A in Proteus.

```

1  #include <main.h>
2
3  /* Main.c file for Project X
4   * Created: Tue May 6 2025
5   * Processor: PIC16F877A
6   * Compiler: CCS for PIC
7   */
8
9  #include <16F877A.h>
10 #use delay(clock=20000000) // 20 MHz crystal
11 #use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, ERRORS) // UART config
12
13 void main() {
14     char receivedData;
15
16     // Print initialization message
17     printf("UART Initialized\n\r");
18
19     while(TRUE) {
20         // Transmit data every 500ms
21         printf("Transmit Hello World\n\r");
22         delay_ms(500);
23
24         // Check for incoming data and echo it back
25         if(kbhit()) {
26             receivedData = getc(); // Read the incoming character
27             printf("Received data: [%c]\n\r", receivedData); // Print received data
28         }
29
30         delay_ms(500);
31     }
32 }
33

```

Figure 18 UART transmission code for the PIC 16F877A in CCS.

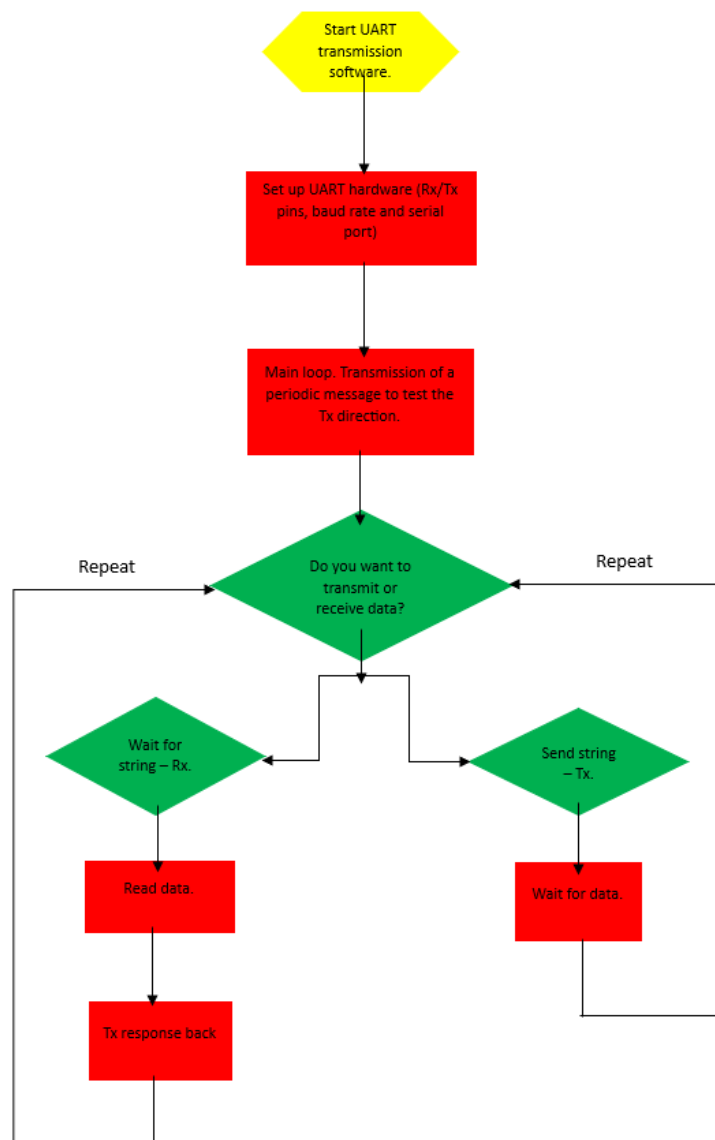


Figure 19 UART transmission test software flowchart.

Hardware development of circuit and PCB design in KiCAD

The hardware development involved designing using KiCAD and having the PCBs manufactured by an external company in industry. The circuit shows the voltage and protocol conversion of RS-232 to TTL voltages either 3.3v or 5.0v between serial UART and TCP protocols. The connectors on each end of the circuit are DB9 serial connector on the RS232 side and an eight-pin header on the TTL connector on the TTL side for the serial UART signals: TX_OUT, RX_IN, RTS_OUT, CTS_IN with alternating GND connections. There are four UART channels per PCB each with a single MAX232E+ integrated voltage converter (I1, I2, I3 & I4) where each MAX232E+ IC connects to an RS232 side and a TTL side to invert and shift RS232 voltages ($\pm 12v$) to TTL voltages. The overall KiCAD circuit diagram for the system is shown in Figure 20 showing the RS232 to TTL voltage translation for four serial UART channels.

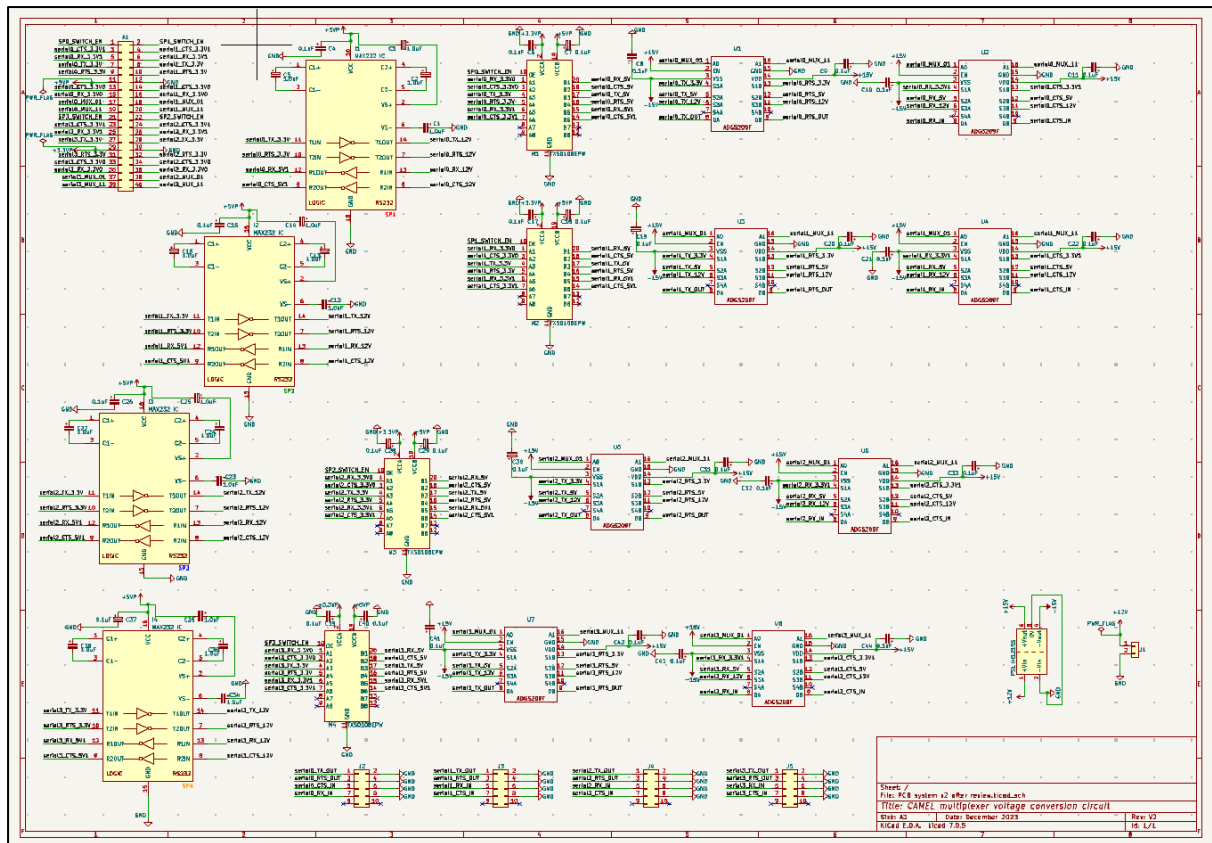


Figure 20 KiCAD circuit diagram.

The MAX232E+ integrated circuit component is designed to change voltage levels between TTL logic (0 – 5v & 0 – 3.3v) and the RS-232 serial standard. The MAX232E+ integrated circuit is designed to translate voltage levels between TTL logic (0–5v) and the RS-232 standard (typically $\pm 10\text{v}$ to $\pm 12\text{v}$). However, the IC itself operates from a single 5v power supply, which is insufficient to directly generate the necessary RS-232 voltage levels. To solve this problem, the MAX232E+ integrated circuit component utilises an internal charge pump circuit that uses external capacitors to generate the higher positive and negative voltages needed. The charge pump works in two main phases which are voltage doubling and voltage inversion. Initially, the first one set of capacitors doubles the supply voltage, forming an intermediate voltage of approximately +10v. Then, another set of capacitors inverts this voltage, generating approximately –10v, which is important because RS-232 signalling requires both positive and negative voltages relative to ground. The capacitors are first connected in parallel to the supply voltage, allowing them to charge to the same voltage. The MAX232 IC's internal logic then switches the capacitors to be connected in series so this doubles or inverts the voltage. The first charge pump doubles the input voltage, and the second inverts it, resulting in +10V and -10V outputs from a single 5V supply. The external capacitors connected to the MAX232E+ have several purposes for example, during each phase of the charge pump operation, capacitors store and transfer electrical charge necessary for boosting and inverting the voltage. After voltage multiplication or inversion, capacitors help smooth out voltage ripples, maintaining a relatively steady DC output suitable for reliable RS-232 communication. The MAX232E+ voltage translator requires four capacitors connected for the charge pump stages and one additional capacitor to stabilize the IC's internal voltage reference.

In the RS-232 to TTL level shifting circuit an ADG5209F analogue multiplexer per UART channel is incorporated and several functions. The ADG5209F is a high-voltage, fault-protected, dual 4:1

analogue multiplexer, meaning it internally contains two independent multiplexers, each capable of routing one of four inputs to a single output under digital control. Within this context of RS-232 to TTL conversion, the multiplexer is connected between the UART transceiver (such as the MAX232) and the microcontroller or system's TTL-level UART interface. The primary purpose of using the ADG5209F in this design is to selectively route UART signals based on system conditions. A second important purpose is fault protection because the ADG5209F features overvoltage tolerance up to $\pm 55\text{V}$ on its analogue pins. Regardless of whether its supply voltage is present which is critical in RS-232 communication, where external connections might expose the UART lines to transient high voltages and electrostatic discharge (ESD) and by placing the ADG5209F between the external RS-232 transceiver and the internal TTL domain, the circuit protects sensitive microcontroller or FPGA input pins from electrical damage. Therefore, the multiplexer acts as an electrical firewall in addition to a routing device.

The TX0108EPW in the RS-232 to TTL to TCP/IP interface circuit is used to translate voltage levels bidirectionally between the UART and the TCP module, protect devices from overvoltage, preserve signal integrity at high speeds, manage power sequencing safely. UART interfaces involved in TCP/IP communication often operate at relatively high baud rates like 115200 bps, 921600 bps, or higher, especially when a bridge to a fast Ethernet or Wi-Fi link. The TX0108EPW is designed for high-speed operation, capable of supporting signal transitions up to 60 Mbps (megabits per second) with low propagation delay ($\sim 4\text{ ns}$ typical). This ensures that even at high baud rates, the level shifting does not introduce asynchronous timing errors or noise that could corrupt UART frames, especially over longer PCB traces. The circuit requires a voltage regulator in the system because RS-232 driver signalling circuits require significant positive and negative voltage switching ranges typically between $+12\text{v}$ and -12v . Modern RS-232 transceivers like the MAX232E+ voltage converter internally generate these voltage switching ranges from a smaller supply (e.g., 5V), using a charge pump that generates voltages of approximately $\pm 7\text{v}$ to $\pm 10\text{v}$. Figure x is the 4 layer KiCAD PCB diagram which has two planes on the top and bottom for UART signals and the two middle planes where one is a GND and the other is the power plane for the 12v, 5.0v and 3.3v power traces.

RS232 communication typically operates with voltage levels in the range of $\pm 12\text{V}$ for data transmission, where $+12\text{V}$ represents a logical "1" and -12V represents a logical "0". In contrast, TTL (Transistor-Transistor Logic) uses much lower voltage levels, typically 0V for a logical "0" and either 3.3V or 5V for a logical "1". To ensure proper communication between RS232 and TTL devices, it's essential to convert the higher RS232 voltage levels to the lower TTL levels. When regulating directly from a higher voltage like 15V to 5V or 3.3V, a linear regulator would have to drop a significant amount of voltage (e.g., 15V to 5V results in a 10V drop). This excess voltage difference translates into wasted power as heat. However, by inserting a 12V rail, the voltage drop is much smaller (e.g., from 15V to 12V, which is only a 3V drop), making the process more efficient and reducing heat dissipation. For instance, the power loss is much less when regulating from 12V compared to regulating directly from 15V. A 15V to 12V regulator first reduces the input voltage from 15V to 12V before it is supplied to the RS232 to TTL conversion circuitry. This step provides a more suitable and stable input voltage for the converter. By doing so, it helps manage the voltage difference between the 12V input and the final TTL output (such as 5V or 3.3V), making the conversion process more manageable and reducing the chances of excessive power loss or heat buildup.

If the system requires 3.3V or 5V for the UART, microcontrollers, or TCP/IP modules, and you're regulating directly from 15V, the voltage drop across the regulator is significant:
Voltage drop = $15\text{v} - 3.3\text{v} = 11.7\text{v}$

In a linear regulator, this voltage difference is directly multiplied by the load current to find the wasted power: $P_{\text{loss}} = (V_{\text{in}} - V_{\text{out}}) \times I_{\text{load}}$

Example: If the TCP/IP module draws 150 mA, then: $P_{\text{loss}} = 11.7\text{v} \times 0.15\text{A} = 1.755\text{W}$

By connecting a 12V rail before the linear regulator, the voltage drops to 5V or 3.3V becomes smaller. For example: Voltage drop = $12\text{v} - 5\text{v} = 7\text{v}$

This results in a much smaller power loss: $P_{\text{loss}} = 7\text{v} \times 0.15\text{A} = 1.05\text{W}$

This is 40% less heating compared to dropping directly from 15V.

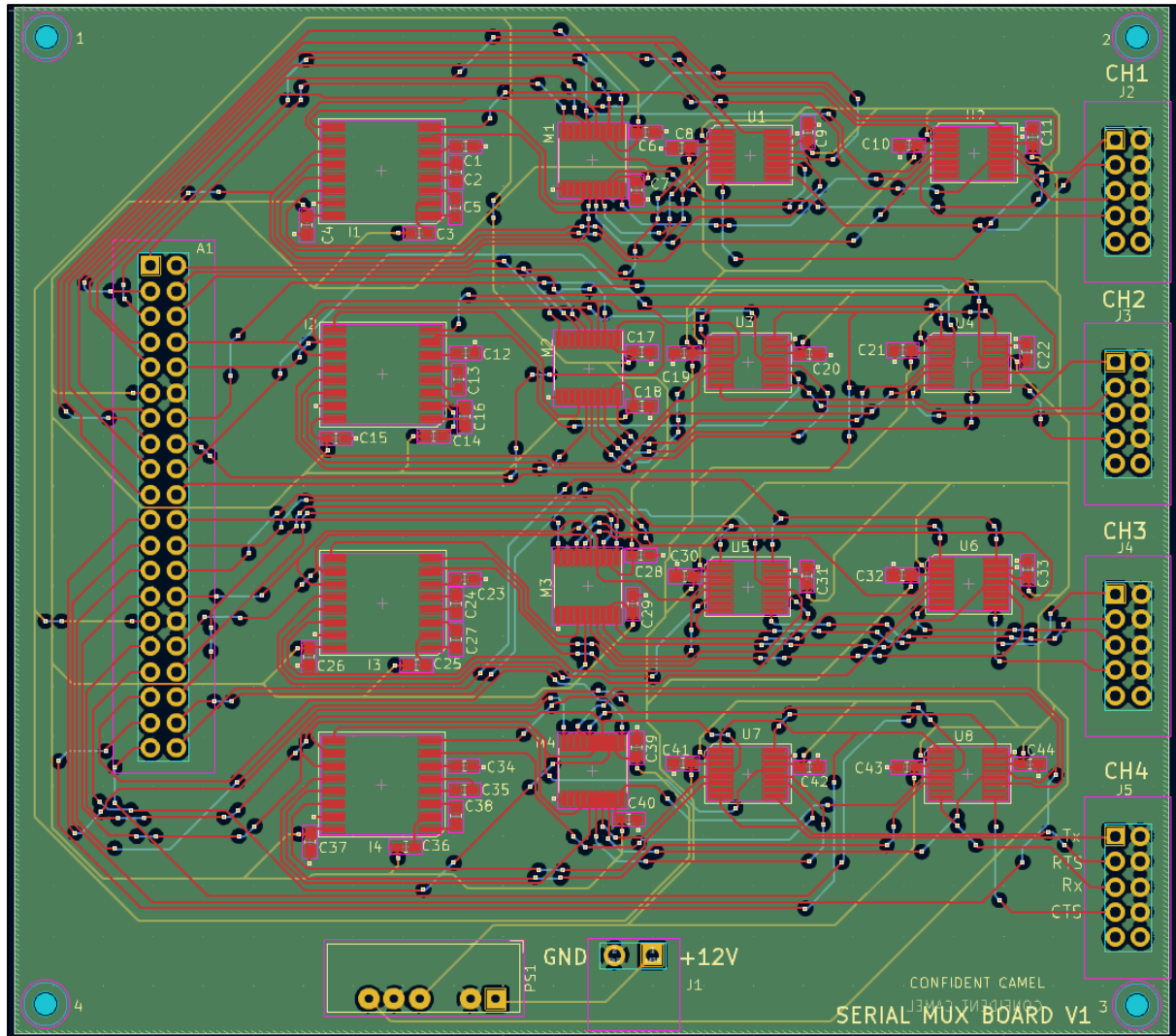


Figure 21 KiCAD PCB diagram.

In this circuit where analogue multiplexers are implemented, decoupling capacitors have been connected close to the power supply pins. The decoupling capacitors connected close to the power supply pins of the ADG2509F analogue 4:1 dual multiplexer for local energy storage because when the internal switches of the ADG5209F multiplexer change state (for example, when a digital control input selects a different analogue channel), there is a sudden increase of current due to charging and discharging of internal gate capacitances and load capacitances. The decoupling capacitors act as local charge reservoirs, providing the instantaneous current required during these transient events and without these capacitors, the supply voltage could briefly spike, potentially causing improper logic thresholds and timing errors of the asynchronous UART protocol. The decoupling capacitors also act as noise suppressors through high-frequency filtering because digital switching noise from

the control signals or from adjacent circuitry (such as microcontrollers, FPGAs, or RS-232 drivers) can couple into the power supply lines. This noise can propagate into the analogue switching circuitry inside the ADG5209F, degrading the multiplexer's performance, causing unwanted crosstalk between channels, or introducing jitter and distortion on the UART signals. By placing small, high-frequency capacitors (typically 100 nF ceramic capacitors) close to the supply pins, high-frequency noise is shorted to ground locally, preventing it from entering the sensitive internal analogue sections of the multiplexer. Figures 22 and 23 are photos of the hardware with a single serial channel soldered up with components and the offboard connectors.

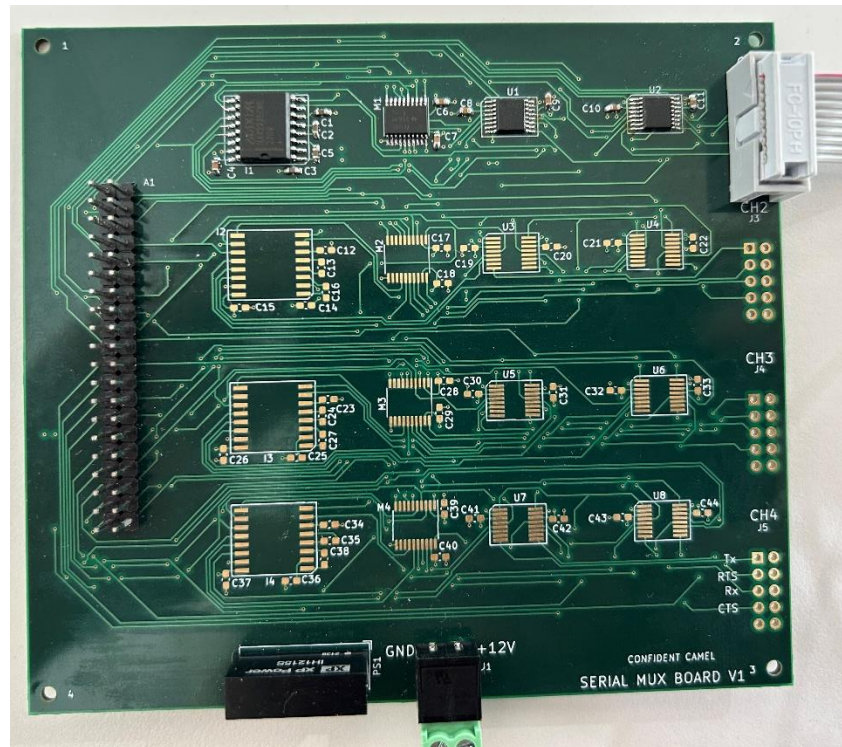


Figure 22 Hardware photo.

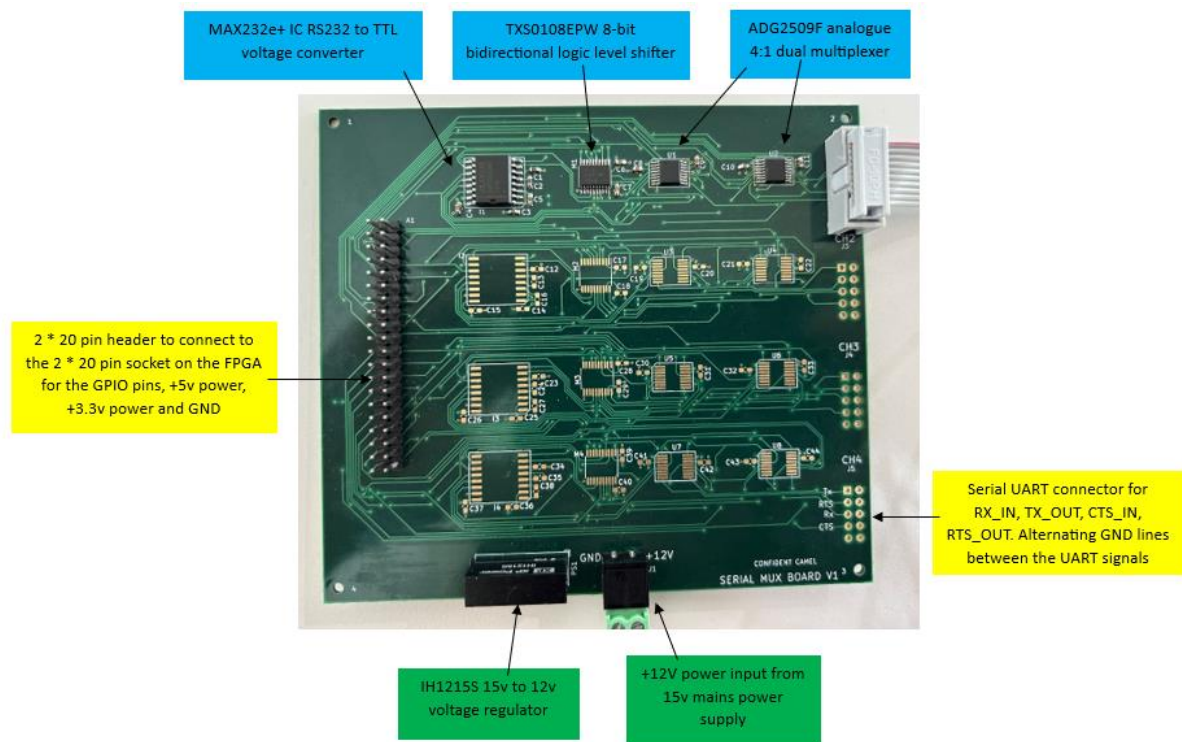


Figure 23 Hardware photo with component and connector annotation.

Software development

The software for the system was developed in three distinct sections, the first of which is the Serial to TCP bridge and configuration of ports software written in Python. This script is a serial-to-TCP bridge system using the Flask web framework, enabling remote configuration and control of up to four UART serial ports, with optional GPIO-based hardware control. It supports dynamic port configuration via a REST API and real-time, bidirectional data forwarding between serial ports and TCP clients. The second section is the embedded Linux device tree and kernel developed in Debian using the Bookworm version. The third section is the UART channel interface for the four channels per FPGA and PCB written in VHDL in Intel Quartus Prime.

Python Serial to TCP bridge and configuration software

This code sets up a system where each UART port on a device can be exposed to a TCP client over the network. Users can change serial settings or switch voltage levels via HTTP POST requests. Behind the scenes, it uses threads and non-blocking I/O to allow simultaneous handling of multiple connections and devices a diagram of which is shown in Figure 26 showing the two threads and the serial dictionary which stores all the configurable and fixed parameters and the three file handles. The main capabilities and features of the bridge.py script is that the software supports 4 serial UART ports that are configurable dynamically through HTTP requests and the users can remotely configure through the UI the: baud rate; TCP_port number; voltage to either 3.3v, 5.0v or 12v and enabling the use of the RTS and CTS lines. It's possible to configure multiple serial ports at once up to four or just a single port before starting the bridge and the transmission of data. A complete flowchart of how the software works is shown in Figure 25 from starting the web server to when the web connection ends. The python code is comprised of several functions and flask routes which are explained below in Figure 24.

Libraries

- Imports Flask for the web interface and REST API.
- Uses serial for UART communication (via pyserial), socket for TCP, select for I/O multiplexing, and gpio (custom or hardware-specific library) for voltage control.

Data Structures for Serial Port Management

Each UART port (serial0 to serial3) has a dictionary storing:

- baud_rate: Serial communication speed.
- tcp_port: TCP port for remote clients.
- rts_cts: Hardware flow control flag.
- serial_device: Linux device (e.g., /dev/ttyAL0).
- voltage: Voltage level for GPIO-controlled interface.
- File handles for serial and TCP sockets.
- A flag configured indicating if the port is currently active.

Python functions and flask routes

Function name	Purpose
Configure_ports()	Opens serial ports and TCP listening sockets as needed, then starts a background thread to monitor data.
Close_all_file_handles()	Cleanly close all open serial and TCP file handles.
Handle_ports()	Background thread: Manages serial ↔ TCP data transfer and client handling. Uses select. select() to monitor all active sockets and serial ports. For each port: • If data comes from serial: send to TCP client. • If data comes from TCP client: write to serial. • If new TCP client connects then it accepts and store client handle or rejects if the client is already busy. • This function loops indefinitely, acting as the communication core.
reconfigure_port()	when a new baud rate or TCP port is received, closing the old ones, and calling configure_ports() again.
setup_gpio_for_ports()	GPIO initialise and voltage switching for external hardware multiplexing.

Route	Method	Purpose
/	GET	Serves the index.html UI.
/configure_port	POST	Accepts JSON payload to update the port config with baud rate and TCP port

Figure 24 Python functions and flask routes.

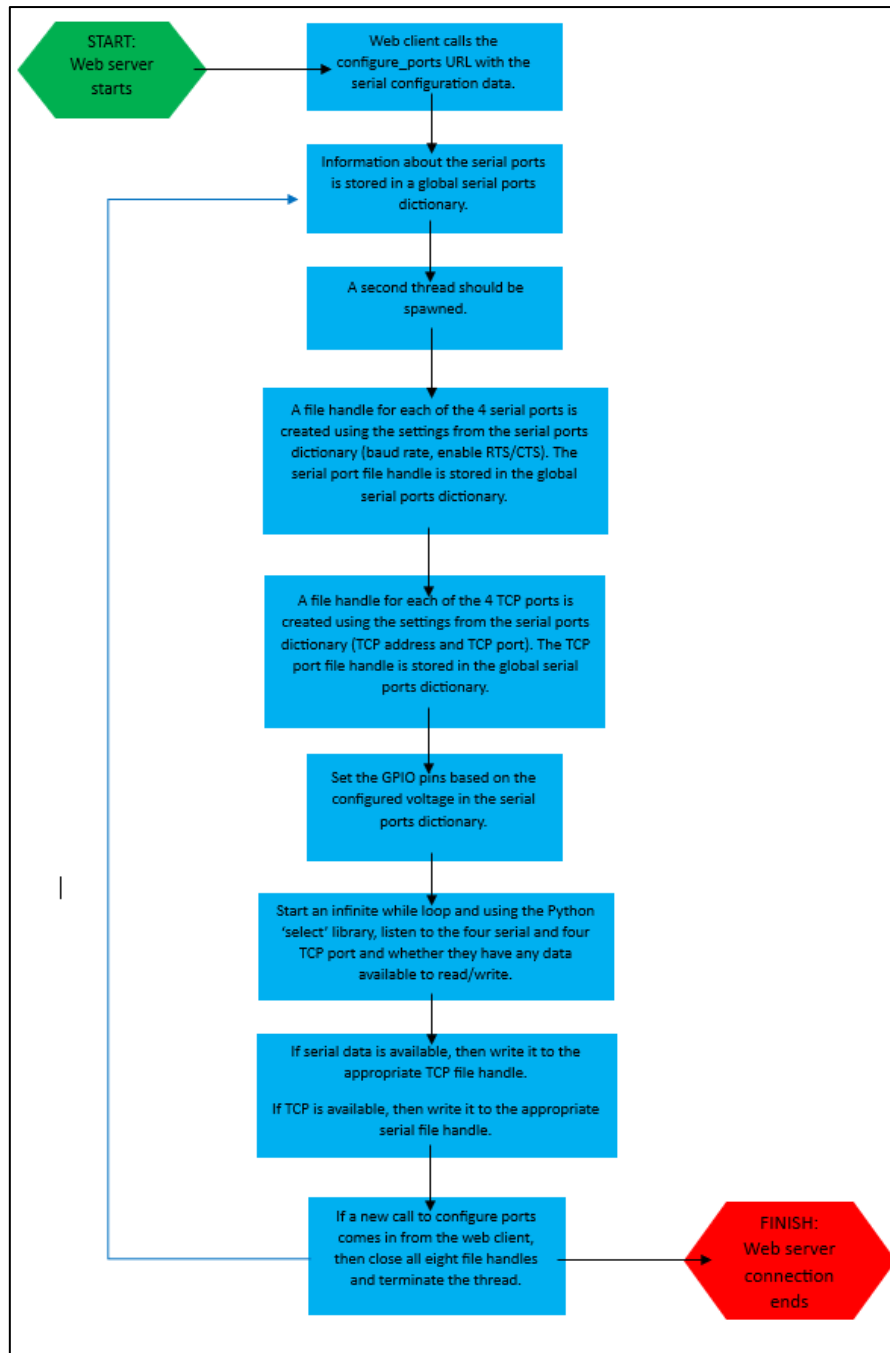


Figure 25 Python bridge.py software flowchart.

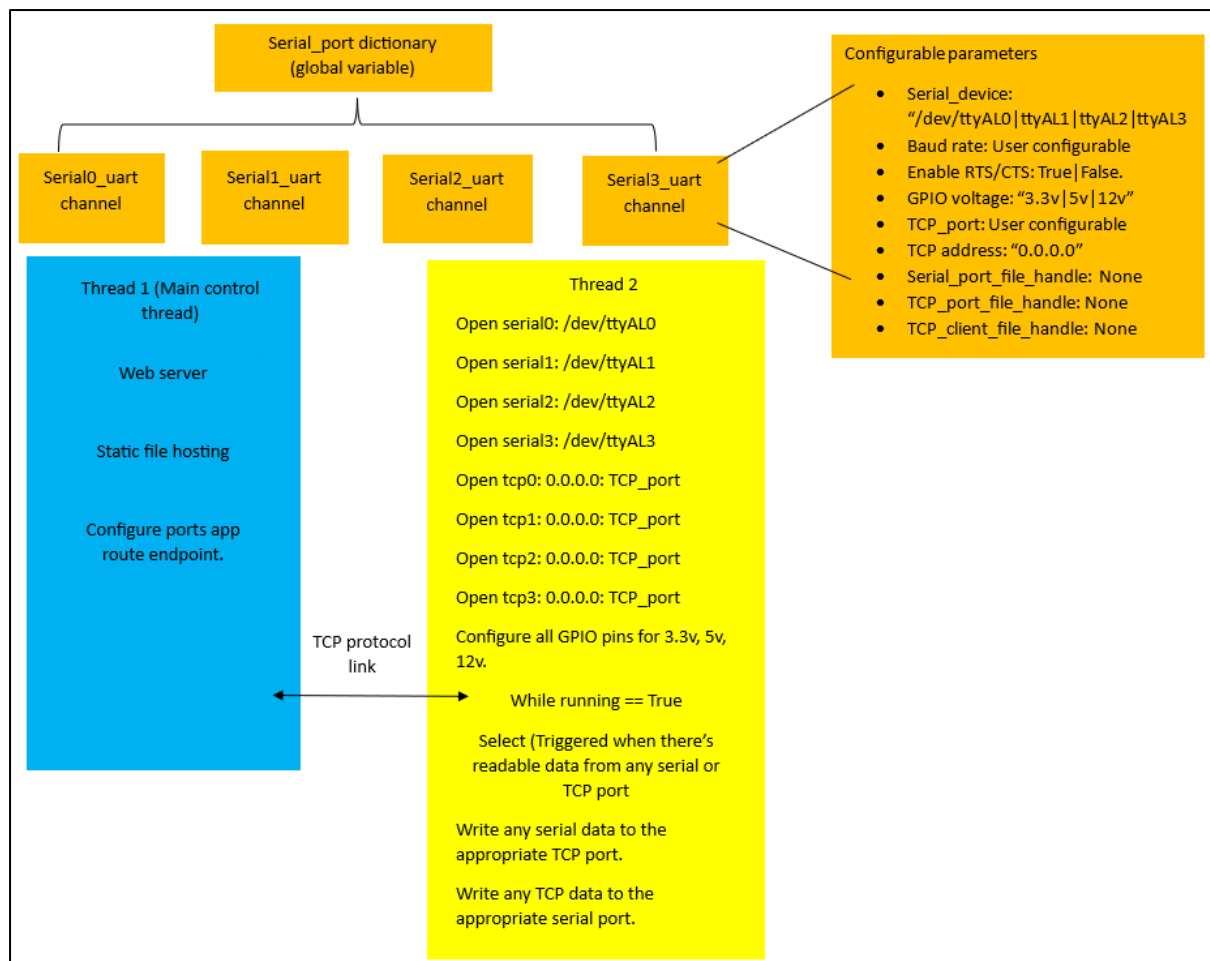


Figure 26 Python bridge.py thread and global dictionary structure and TCP protocol link.

The UI was developed using the flask framework in Python and an example of a configured serial port is shown in Figure 27. The serial to TCP bridge is bidirectional and can transmit and receive serial or TCP data between the port and the client depending on the user requirement and the configuration of the port is sent on the HTTP connection to the server.

Serial Port Configuration

Select Port: serial0_uart (configured) ▼ (configured)

TCP Address: 0.0.0.0 (fixed)

TCP Port: 50000 ▼

Baud Rate: 115200 ▼

Voltage: 3.3V ▼

Data Bits: 8 (fixed)

Stop Bits: 1 (fixed)

RTS/CTS: ☐

Configure Ports

Figure 27 UI example for serial port configuration.

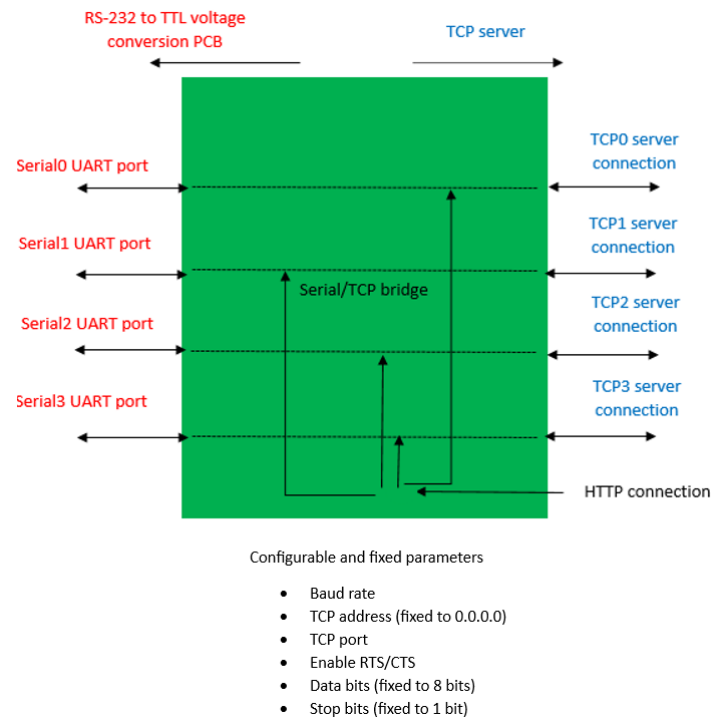


Figure 28 Serial port and TCP server connection showing data flow directions of serial UART and TCP data.

Bookworm Debian Embedded Linux OS device tree and kernel

A device tree is a hierarchical data structure utilised by the Linux kernel to describe the hardware layout of the system. This part of the device tree specific to the system being developed defines UART peripherals and one GPIO controller in the FPGA system, using the Altera 16550 UART and PIO IP cores. The kernel will use this information to bind appropriate drivers, map memory addresses, configure interrupts and handle peripheral interactions such as serial communications and GPIO control. Fpga_UART0 - Fpga_UART3 defines the nodes for the four UART peripherals using 16550 compatible UARTs with FIFO-128 support using the serial_8250 driver. The FIFO size of 128 informs the kernel's serial driver 8250 that this UART has 128-byte FIFOs for both the transmission and receive of data. The fpga_pio0 nodes describes the GPIO controller with 12 GPIO pins that can be accessed and controlled by the Linux kernel.

Field	Description
Fpga_UARTY@0XC0000X00	Fpga_UARTY is the label where Y=0,1,2,3 AND 0XC0000X00 represents the address of the UART peripheral. The @ symbol uniquely identifies each node in the address space of the device.
Compatible = "altr,16550-FIFO128"	A 16550 – compatible UART with a 128-byte FIFO for nodes UART0 – UART3
Reg = <0XC0000X00 0x200	Base memory address 0XC0000X00 where X = 0,2,4 or 6 and Size = 0x200 or 512 bytes
Interrupts = <72 73 74 75>	Unique interrupt handlers from 72-75 for the UART nodes UART0 – UART3 connected
Clock-frequency = <50000000>	50MHz input clock
Reg-shift = <2>	Register spacing multiplier in this UART interface register are spaced every 4 bytes (2 ²)
Fifo-size = <128>	The size of the internal buffer (FIFO) for sending/receiving data and bound to the 8250 serial driver
Fpga_pio0@C0000800	Unit address at 0xC0000800. FPGA mapped IO port
Compatible = "altr,pio-1.0"	Indicates that the PIO core is an Altera specific implementation, and the kernel should use the Altera GPIO driver.
Reg = <0xC0000800 0x20>	32 byte memory space and size of the <u>pio</u>
Altr,ngpio = <12>	This GPIO controller provides 12 GPIO lines
#gpio-cells = <2>	This defines how GPIOs are referenced elsewhere in the device tree. Two cells: first is GPIO number, second is flags (e.g. 0 = active high, 1 = active low)

Figure 29 Explanation of the device tree for the interfaces and GPIO (the code is shown in Figure 30).

Interactions between the device tree and the Linux kernel

- Driver binding
 - The kernel uses the compatible = "altr,16550-FIFO128" property to load the appropriate UART driver (in this case serial_8250 with FIFO support). The kernel then registers the UART device under /dev/ttySx for serial communication.
 - The gpio-altera driver binds to the compatible = "altr,pio-1.0" property and initialises the GPIO controller.
- Interrupts
 - The interrupts property informs the kernel about the interrupt number (72 = fpga_UART0). The kernel configures the interrupt controller and registers an interrupt service routine to handle interrupts triggered by UART events (e.g. data available in the Rx FIFO).
- Memory mapped IO
 - The kernel utilises the reg property to access the memory mapped IO regions of the peripherals. For the UART interface, the memory region starts at 0xc0000600 and is 512 bytes wide and for the GPIO the region starts at 0xc0000800 and is 32 bytes wide.
- Clocks
 - The kernel can use the clock frequency property to configure the clock source for the UART peripheral.

The device tree in embedded Linux specific to this system for the four UART interfaces and one GPIO controller is shown in Figure 30 and Figure 31 shows the memory mapped addresses and interrupts for the four UART channels 0 – 3.

```

fpga_uart0: fpga_uart0@C0000600
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000600 0x200>;
    interrupts = <72>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart1: fpga_uart1@C0000400
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000400 0x200>;
    interrupts = <73>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart2: fpga_uart2@C0000200
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000200 0x200>;
    interrupts = <74>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart3: fpga_uart3@C0000000
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000000 0x200>;
    interrupts = <75>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_pio0: fpga_pio0@C0000800
{
    compatible = "altr,pio-1.0";
    reg = <0xC0000800 0x20>;
    altr,ngpio=<12>;
    #gpio-cells=<2>;
    gpio-controller;
};

```

Figure 30 Device tree source code for the FPGA UART and GPIO interface.

UART	Base address	IRQ	Size
UART0	0xC0000600	72	0x200
UART1	0xC0000400	73	0x200
UART2	0xC0000200	74	0x200
UART3	0xC0000000	75	0x200

Figure 31 UART memory map summary.

UART channel interface in VHDL

This first section of the VHDL defines the signal declarations for four UART interfaces UART0 – UART3.

```

-- UART 1
signal SP1_switch_en      : out std_logic;
signal serial1_mux_01     : out std_logic;
signal serial1_mux_11     : out std_logic;
signal serial1_TX_3_3V    : out std_logic;
signal serial1_RTS_3_3V   : out std_logic;
signal serial1_CTS_3_3V0 : in  std_logic;
signal serial1_RX_3_3V0   : in  std_logic;
signal serial1_CTS_3_3V1 : in  std_logic;
signal serial1_RX_3_3V1   : in  std_logic;

-- UART 2
signal SP2_switch_en      : out std_logic;
signal serial2_mux_01     : out std_logic;
signal serial2_mux_11     : out std_logic;
signal serial2_TX_3_3V    : out std_logic;
signal serial2_RTS_3_3V   : out std_logic;
signal serial2_CTS_3_3V0 : in  std_logic;
signal serial2_RX_3_3V0   : in  std_logic;
signal serial2_CTS_3_3V1 : in  std_logic;
signal serial2_RX_3_3V1   : in  std_logic;

-- UART 3
signal SP3_switch_en      : out std_logic;
signal serial3_mux_01     : out std_logic;
signal serial3_mux_11     : out std_logic;
signal serial3_TX_3_3V    : out std_logic;
signal serial3_RTS_3_3V   : out std_logic;
signal serial3_CTS_3_3V0 : in  std_logic;
signal serial3_RX_3_3V0   : in  std_logic;
signal serial3_CTS_3_3V1 : in  std_logic;
signal serial3_RX_3_3V1   : in  std_logic
);
end de10_nano_clean;

architecture behavioral of de10_nano_clean is

    signal reset : std_logic;
    signal gpo_0 : std_logic_vector(11 downto 0);
    signal serial0_RX_combined : std_logic;
    signal serial0_CTS_combined : std_logic;
    signal serial1_RX_combined : std_logic;
    signal serial1_CTS_combined : std_logic;
    signal serial2_RX_combined : std_logic;
    signal serial2_CTS_combined : std_logic;
    signal serial3_RX_combined : std_logic;
    signal serial3_CTS_combined : std_logic;

```

Figure 32 VHDL signal declaration and combing signals.

Each UART block contains:

Outputs (from the UART block to external hardware)

- SPx_switch_en: enables the switch for selecting the serial port.
- Serialx_mux_01, serialx_mux_11: Control signals for a multiplexer selecting the source/destination for serial communication.
- Serialx_TX_3_3V: UART transmit line at 3.3v logic level.
- Serialx_RTS_3_3V: Ready to send signal at 3.3v

Inputs (to the UART block)

- Serialx_CTS_3_3V0, serialx_RX_3_3V0: Clear to send and receive signals from one source (channel 0).
- Serialx_CTS_3_3V1, serialx_RX_3_3V1: Clear to send and receive signals from one source (channel 1).

The second section of VHDL supports the mechanism for combing the input signals from dual sources for each UART for the multiplexer control logic. The combined signals are input to the UART logic and are dynamically assigned based on the selected channel from the multiplexer.

Breakdown:

- Reset: standard system reset signal.
- Gpo_0: A 12 bit general purpose output signal (e.g. to control multiplexers and enable switches).
- Serialx_RX_COMBINED/serialx_CTS_COMBINED: These represent the selected or combined version of the two available RX and CTS signals (_3_3V0 and _3_3V1) per UART.

The third section of the VHDL defines the top-level IO ports to interface the FPGA and these ports map the UART signals to actual physical pins.

Breakdown:

- Uart_x_rxd: Receive data input from the system
- Uart_x_txd: Transmit data output from the system
- Uart_x_cts_n: Clear to send, active low input signal
- Uart_x_rts_n: Ready to send, active low output signal (_n indicates the signals are active low)
- Gpo_export: External version of the internal gpo_0 signal and can drive control signals for all four UARTs like SPx_switch_en, serialx_mux_01 and serialx_mux_11.

The fourth section of the VHDL is the component instantiation mapping in the top-level design that connects signals from the internal logic to an instantiated module which is the HPS connected UART bridge from Intel/Altera's Qsys Platform Designer. Both these sections of VHDL are shown directly below.

Breakdown:

- Reset line: hps_0_h2f_reset_reset_n mapped to internal reset.
- UART interfaces 0-3: Connecting internal signals to the corresponding UART_X_*ports.
- GPIO control vector: gpo_0_export mapped to internal gpo_0.

```

hps_io_hps_io_spi1_inst_CLK : out std_logic;
hps_io_hps_io_spi1_inst_MOSI : out std_logic;
hps_io_hps_io_spi1_inst_MISO : in std_logic := 'X';
hps_io_hps_io_spi1_inst_SS0 : out std_logic;
hps_io_hps_io_uart0_inst_RX : in std_logic := 'X';
hps_io_hps_io_uart0_inst_TX : out std_logic;
hps_io_hps_io_i2c0_inst_SDA : inout std_logic := 'X';
hps_io_hps_io_i2c0_inst_SCL : inout std_logic := 'X';
hps_io_hps_io_i2c1_inst_SDA : inout std_logic := 'X';
hps_io_hps_io_i2c1_inst_SCL : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO09 : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO35 : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO40 : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO53 : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO54 : inout std_logic := 'X';
hps_io_hps_io_gpio_inst_GPIO61 : inout std_logic := 'X';
hps_0_h2f_reset_reset_n : out std_logic;
uart_0_rxd : in std_logic := 'X';
uart_0_txd : out std_logic;
uart_0_cts_n : in std_logic := 'X';
uart_0_rts_n : out std_logic;
uart_1_rxd : in std_logic := 'X';
uart_1_txd : out std_logic;
uart_1_cts_n : in std_logic := 'X';
uart_1_rts_n : out std_logic;
uart_2_rxd : in std_logic := 'X';
uart_2_txd : out std_logic;
uart_2_cts_n : in std_logic := 'X';
uart_2_rts_n : out std_logic;
uart_3_rxd : in std_logic := 'X';
uart_3_txd : out std_logic;
uart_3_cts_n : in std_logic := 'X';
uart_3_rts_n : out std_logic;
gpo_0_export : out std_logic_vector(11 downto 0);
);
end component de10_nano_clean_soc;

```

Figure 33 Component instantiation mapping.

The final VHDL section is shown below and subsequently explained. It defines how certain GPOS are implemented to control switches and multiplexers for the serial communication interfaces in a system and the logic also deals with the level shifting for RX and CTS signals which are important in serial communication. The gpo_0 signal is a vector of gpo lines indexed from 0-11 and control the enabling of switches and selecting of multiplexer lines.

Switch enable lines example: `SP0_switch_en <= gpo_0(0);` --SP0_switch_en GP0 line.

These lines control the switch enable signals for the four serial ports (0,1,2 & 3) and each switch is enabled or disabled depending on the state of corresponding GP0 line (gpo_0(0) through gpo_0(3)).

- If the line is high (1) then the switch is enabled and if its low (0) then its disabled.

Multiplexer control lines examples: `serial0_mux_01 <= gpo_0(4);` --serial0_mux_01 gpo line.

`serial0_mux_11 <= gpo_0(5);` --serial0_mux_11 gpo line.

These lines control the MUX settings for the four serial ports 0-3. Multiplexers (mux_01 and mux_11) can select different signals for the serial port between the lines Rx, Tx, RTS and CTS.

Each port has two MUX options:

- Mux_01 controlled by gpo_0(n) (for n in 4 to 11)
- Mux_11 controlled by gpo_0(n+1)

Combining signals based on MUX and switch states example:

`Serial0_RX_combined <= serial0_RX_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_RX_3_3V1;`

`Serial0_CTS_combined <= serial0_CTS_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_CTS_3_3V1;`

The RX and CTS signals are combined into a single signal based on the state of the switch enable and MUX lines and this is dependent for each serial port on the state of the corresponding GPO lines that control the switches and multiplexers.

- Serial0_RX_combined is assigned to serial0_RX_3_3V0 (a signal at 3.3V level) when:
 - SP0_switch_en is low (controlled by gpo_0(0))
 - Serial0_mux_01 is low (controlled by gpo_0(4))
 - Serial0_mux_11 is low (controlled by gpo_0(5))

If any of these conditions are false, then the combined RX signal will switch to serial0_RX_3_3V1 or a different signal at a different voltage. The same logic explained for the combined RX signal is also true for the combined CTS signal. Figure 32 shows the UART channel schematic from the Intel Quartus Prime.

```

hps_0_h2f_reset_reset_n    => reset,
uart_0_rxd                  => serial0_RX_combined, -- uart_0.rxd
uart_0_txd                  => serial0_TX_3_3V,      -- .txd
uart_0_cts_n                => serial0_CTS_combined, -- .cts_n
uart_0_rts_n                => serial0_RTS_3_3V,     -- .rts_n
uart_1_rxd                  => serial1_RX_Combined,  -- uart_1.rxd
uart_1_txd                  => serial1_TX_3_3V,      -- .txd
uart_1_cts_n                => serial1_CTS_combined, -- .cts_n
uart_1_rts_n                => serial1_RTS_3_3V,     -- .rts_n
uart_2_rxd                  => serial2_RX_combined,  -- uart_2.rxd
uart_2_txd                  => serial2_TX_3_3V,      -- .txd
uart_2_cts_n                => serial2_CTS_combined, -- .cts_n
uart_2_rts_n                => serial2_RTS_3_3V,     -- .rts_n
uart_3_rxd                  => serial3_RX_combined,  -- uart_3.rxd
uart_3_txd                  => serial3_TX_3_3V,      -- .txd
uart_3_cts_n                => serial3_CTS_combined, -- .cts_n
uart_3_rts_n                => serial3_RTS_3_3V,     -- .rts_n
gpo_0_export                => gpo_0               -- gpo_0.export
);

-- Switch_en, MUX_01 and MUX_11 GPOs lines
SP0_switch_en <= gpo_0(0); -- SP0_switch_en GPO line
SP1_switch_en <= gpo_0(1); -- SP1_switch_en GPO line
SP2_switch_en <= gpo_0(2); -- SP2_switch_en GPO line
SP3_switch_en <= gpo_0(3); -- SP3_switch_en GPO line
serial0_mux_01 <= gpo_0(4); -- serial0_MUX_01 GPO line
serial0_mux_11 <= gpo_0(5); -- serial0_MUX_11 GPO line
serial1_mux_01 <= gpo_0(6); -- serial1_MUX_01 GPO line
serial1_mux_11 <= gpo_0(7); -- serial1_MUX_11 GPO line
serial2_mux_01 <= gpo_0(8); -- serial2_MUX_01 GPO line
serial2_mux_11 <= gpo_0(9); -- serial2_MUX_11 GPO line
serial3_mux_01 <= gpo_0(10); -- serial3_MUX_01 GPO line
serial3_mux_11 <= gpo_0(11); -- serial3_MUX_11 GPO line
serial0_RX_combined <= serial0_RX_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_RX_3_3V1;
serial0_CTS_combined <= serial0_CTS_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_CTS_3_3V1;
serial1_RX_combined <= serial1_RX_3_3V0 when (gpo_0(1) = '0' and gpo_0(6) = '0' and gpo_0(7) = '0') else serial1_RX_3_3V1;
serial1_CTS_combined <= serial1_CTS_3_3V0 when (gpo_0(1) = '0' and gpo_0(6) = '0' and gpo_0(7) = '0') else serial1_CTS_3_3V1;
serial2_RX_combined <= serial2_RX_3_3V0 when (gpo_0(2) = '0' and gpo_0(8) = '0' and gpo_0(9) = '0') else serial2_RX_3_3V1;
serial2_CTS_combined <= serial2_CTS_3_3V0 when (gpo_0(2) = '0' and gpo_0(8) = '0' and gpo_0(9) = '0') else serial2_CTS_3_3V1;
serial3_RX_combined <= serial3_RX_3_3V0 when (gpo_0(3) = '0' and gpo_0(10) = '0' and gpo_0(11) = '0') else serial3_RX_3_3V1;
serial3_CTS_combined <= serial3_CTS_3_3V0 when (gpo_0(3) = '0' and gpo_0(10) = '0' and gpo_0(11) = '0') else serial3_CTS_3_3V1;

```

Figure 34 GPO logic control of switches and MUXs.

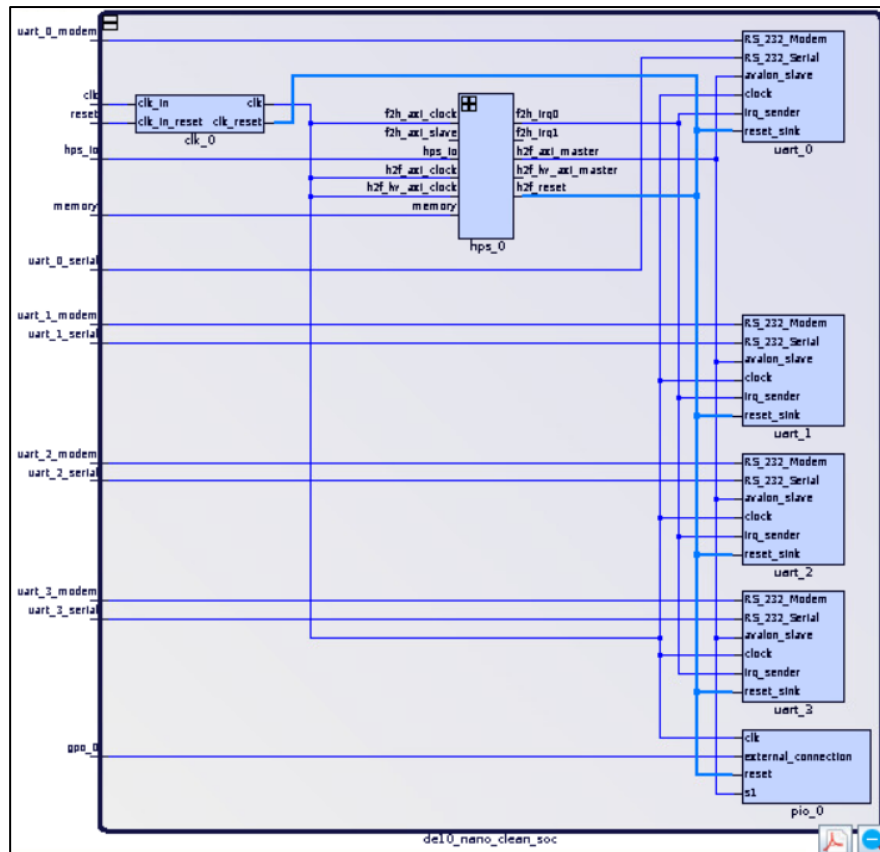


Figure 35 UART schematic diagram from Intel Quartus Prime.

Testing methodology

The software and hardware elements of the project have been extensively tested and documented to prove that the system functions as it is required and successfully meets all the requirements of the project as dictated by the wider project in my organisation who are the end customers of the system. The system has undergone several types of functional and acceptance testing to ensure that all components of the hardware and software function both separately and together as a whole system.

Electrical design rule checking in KiCAD software

The integrity of the circuit schematic was checked for logical and electrical consistency rules using the electrical design rules check feature in KiCAD. The ERC is an important design verification tool that checks the schematic is electrically sound and catch common errors that might lead to design failures or prevent proper PCB layout. The ERC does not simulate circuit behaviour and flow, or voltage/current correctness and it doesn't detect overcurrent, heat issues, or analogue design mistakes. The feature checks for unconnected pins, conflicting pin types, (e.g., output-to-output), missing power connections and floating inputs. Figure 36 shows an initial run of the electrical design rule checker in KiCAD so understand the types of errors predominantly about highlighting power sources and nets. The electrical rules check feature in KiCAD enabled any errors in the circuit to be identified and highlighted so they could be corrected before the PCB was laid out. In Figure 37 it shows there are several errors in the circuit which have been highlighted with red arrows on the schematic and warnings have been shown with orange arrows. Some of the errors were displayed because not all the pins on the TXS0108EPW 8-bit bidirectional logic shifter and the off board 2 * 5 pin connectors were utilised and if a pin is left unconnected without an intentional marker (e.g., No

Connect flag) then the ERC will issue a warning or error. Another error shown was that some of the power pins (e.g. VCC) were not connected to a power net or to a power flag component to indicate in the circuit that those pins and nets are power sources. Figure 38 shows the electrical rules checker feature has successfully checked all the circuit and there are no errors or warnings present in the design.

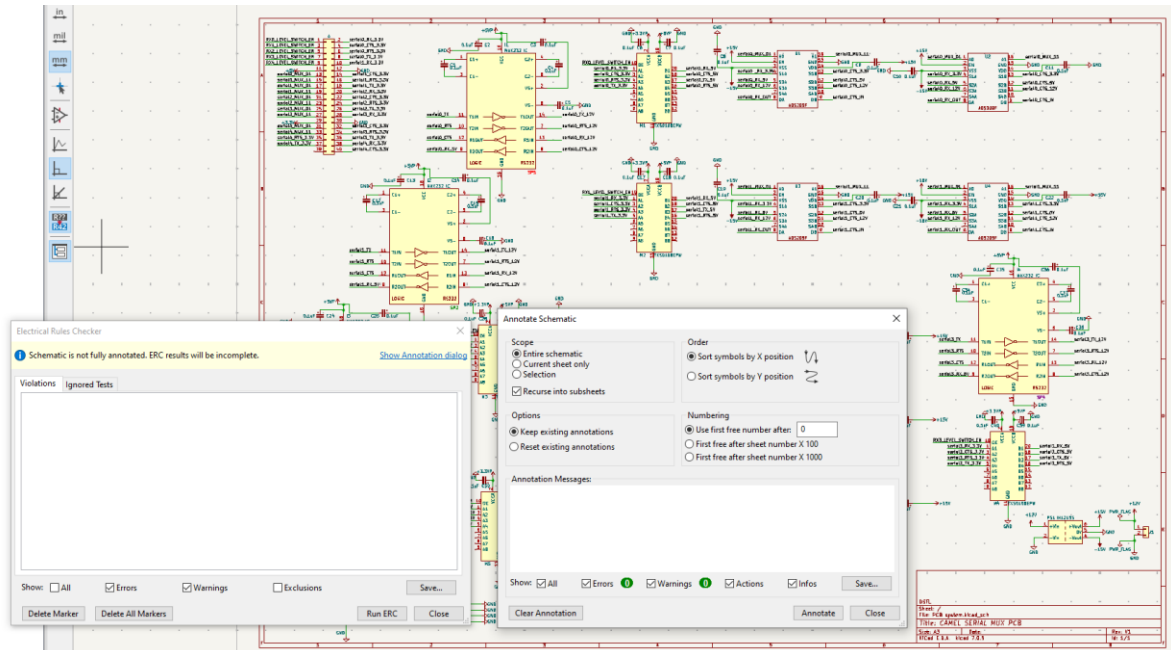


Figure 36 Initial use of the electrical design rules check feature in KiCAD with an incomplete circuit diagram.

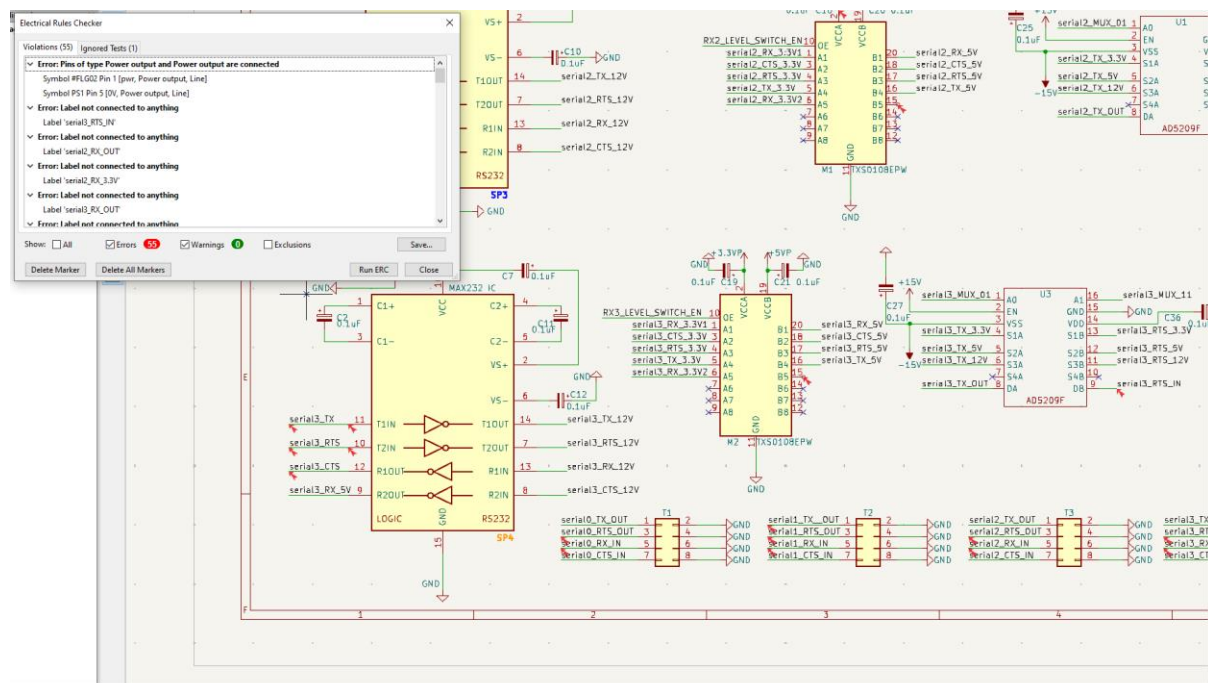


Figure 37 Electrical design rule check feature in KiCAD after an initial run before any errors are corrected.

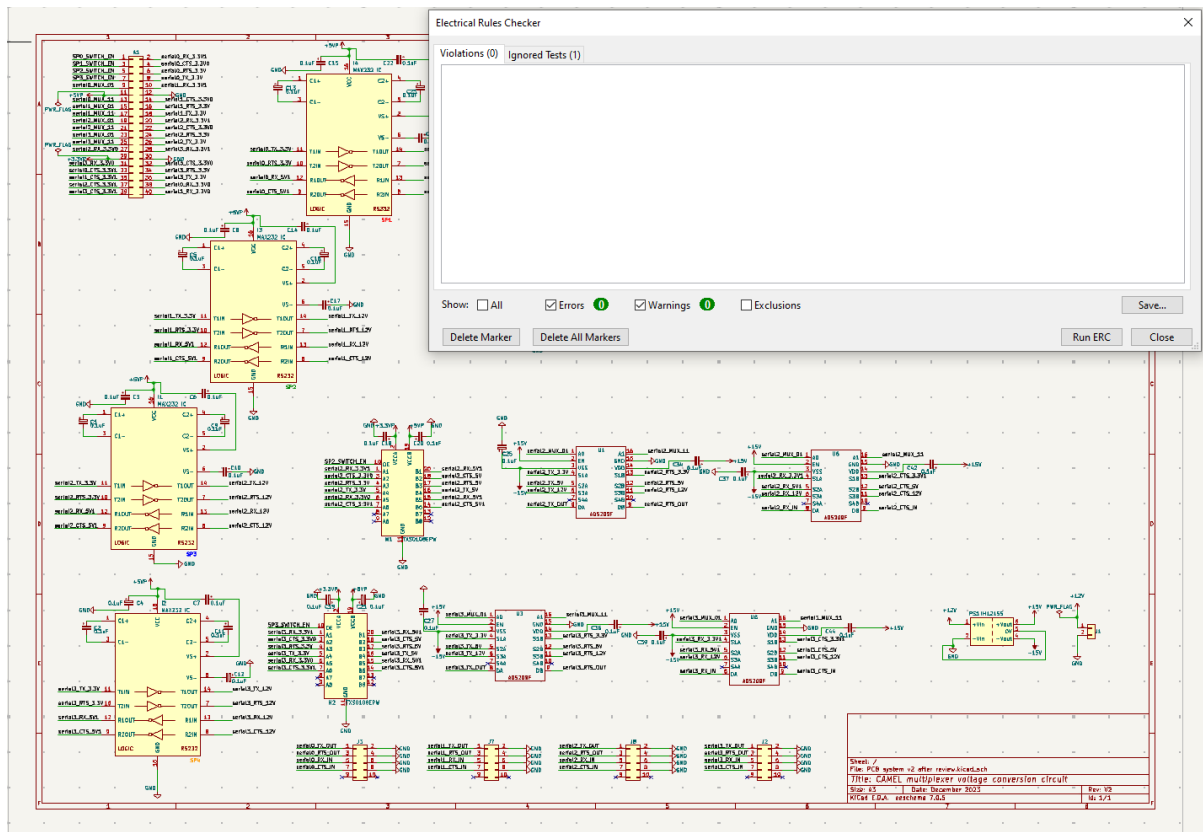


Figure 38 Proof of the circuit passing the electrical design rules check in KiCad.

PCB testing

The PCB was extensively in the lab to verify that the PCB was functioning as required in both Tx and Rx directions and could switch 3.3v, 5.0v and 12v. The guide below shows the testing procedure and results obtained while testing this specific PCB.

Step-by-Step RS232 to TTL PCB Testing Procedure (Lab Guide with Specifics)

1. Visual and Continuity Inspection

Tools: Multimeter and microscope

Checked for:

- Proper orientation of ICs (e.g., MAX232, ADG2509F, TX0108EPW).
- Solder bridges between pins, lifted pads, or missing components.

Verified continuity:

- Vcc to IC V+ pin (e.g., pin 16 on MAX232) = $\sim 0 \Omega$
- GND to IC GND pin (e.g., pin 15 on MAX232) = $\sim 0 \Omega$
- No shorts between Vcc and GND.

2. Power Supply and Idle Current Test

Tools: Suitable 5v power supply

- Connect 5V regulated lab supply to the Vcc pin.
- Current draw:

- A typical MAX232 board should draw 5–15 mA at idle.
- Checked voltages:
 - Vcc: 4.75V–5.25V
 - Internal charge pump output (V+): ~+8V
 - Internal charge pump output (V–): ~-8V
 - These can be probed at the capacitor pins connected to the charge pump.

3. RS232 Loopback Test

- Connect RS232 TX and RX pins together from a single serial channel on the FPGA using a loopback adapter.
- Connect TTL RX and TX to a USB-TTL adapter or microcontroller.
- Send test strings like "Hello" or known byte patterns (e.g., 0x55 = 01010101).
- Using screen in the Linux terminal:
 - Baud rate: 9600, 19200, 115200
 - Data bits: 8, Stop bit: 1, Parity: None
- Pass condition: Sent data is received identically.

4. Oscilloscope Signal Verification

- Tools: 100 MHz oscilloscope and probes
- Measure at:
 - RS232 input (DB9 TX pin 3) — should swing +9V to -9V (typical), up to ± 12 V max.
 - TTL output pin — should swing from 0V (LOW) to 5V (HIGH) or 3.3V, depending on system.
- Measure rise/fall times:
 - TTL: <100 ns
 - RS232: up to 1 μ s (depends on load and cable length)
- Confirm signal integrity:
 - Logic levels:
 - TTL logic HIGH: >2V
 - TTL logic LOW: <0.8V
 - RS232 logic HIGH: < -3V
 - RS232 logic LOW: > +3V

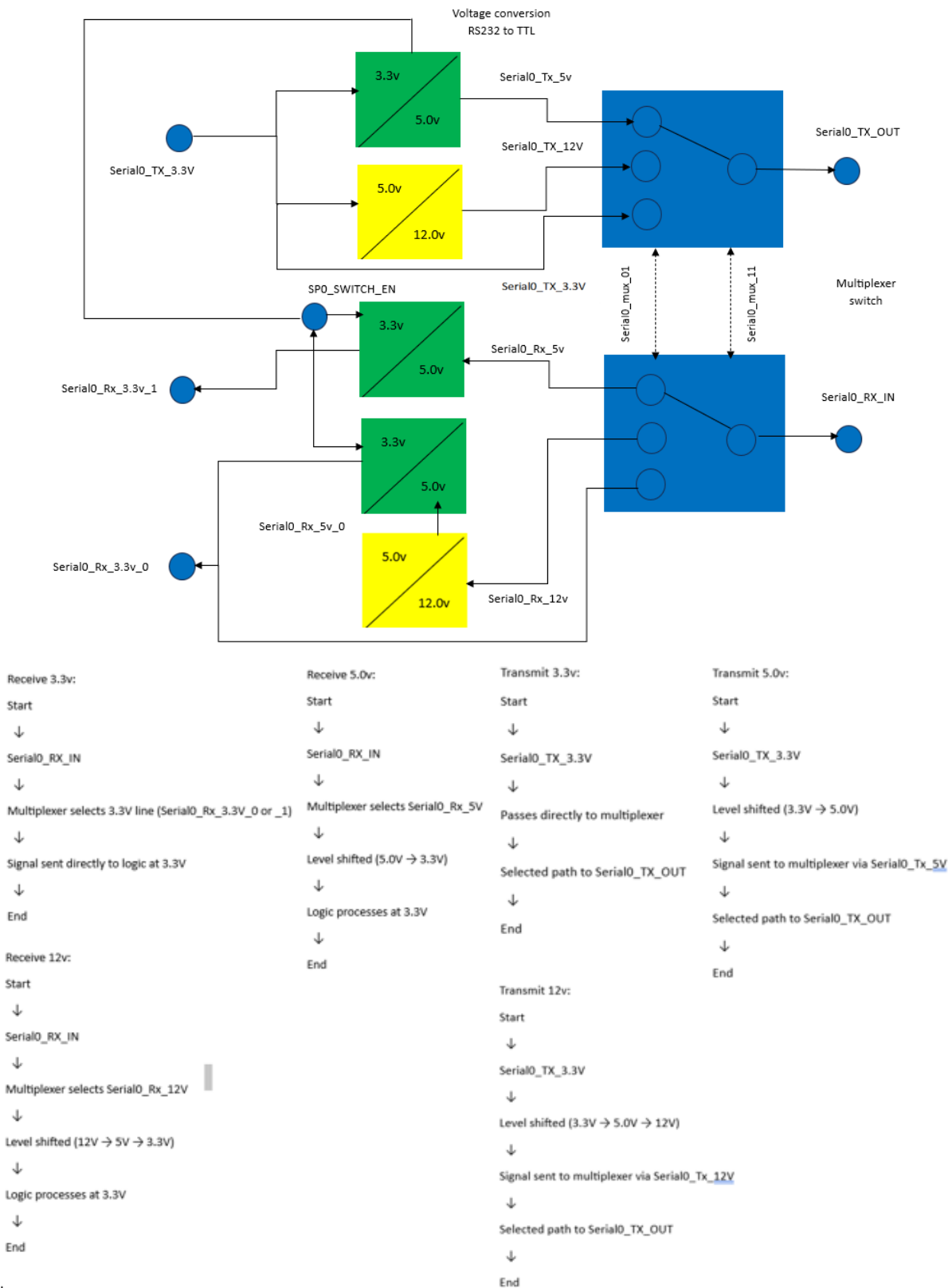


Figure 39 Rx and Tx signal flow for 3.3v, 5.0v and 12v.

GPIO voltage software testing

The systems GPIO voltage switching functionality has been extensively tested in the lab using a logic analyser and oscilloscope to confirm that the GPIO pins can be switched between HIGH and LOW when changing the voltages between 3.3v, 5v or 12v. there are 12 GPIO pins per serial UART channel which are SPX_SWITCH_EN, serialx_CTS_3.3V1, serialx_RX_3.3V1, serialx_RX_3.3V0, serialx_CTS_3.3V0, serialx_TX_3.3V, serialX_RTS_3.3V, serialx_mux_01, serialx_mux_11 where x is the channel number either 0,1,2 or 3. The logic analyser has 12 individual testing probes which all connect to a single oscilloscope input. The logic analyser is an effective tool to test and verify the GPIO pins functionality because it's possible to debug and analyse signals in embedded systems or digital circuit. Multiple digital signals can be observed at the same time allowing all 12 GPIO pins per channel to be monitored at once through the UART communication protocol because the logic analyser has 12 pin connectors. Figure 40 below shows that GPIO pin 10 is functioning appropriately and can be driven HIGH and all the other GPIO pins are LOW. Figure 41 shows that all the GPIO pins in a single selected serial channel are functional and can be driven HIGH or LOW depending on the user requirement and so can be switched between RS232 $\pm 12v$ to 0 – 3.3v or RS232 to 0 – 5.0v.

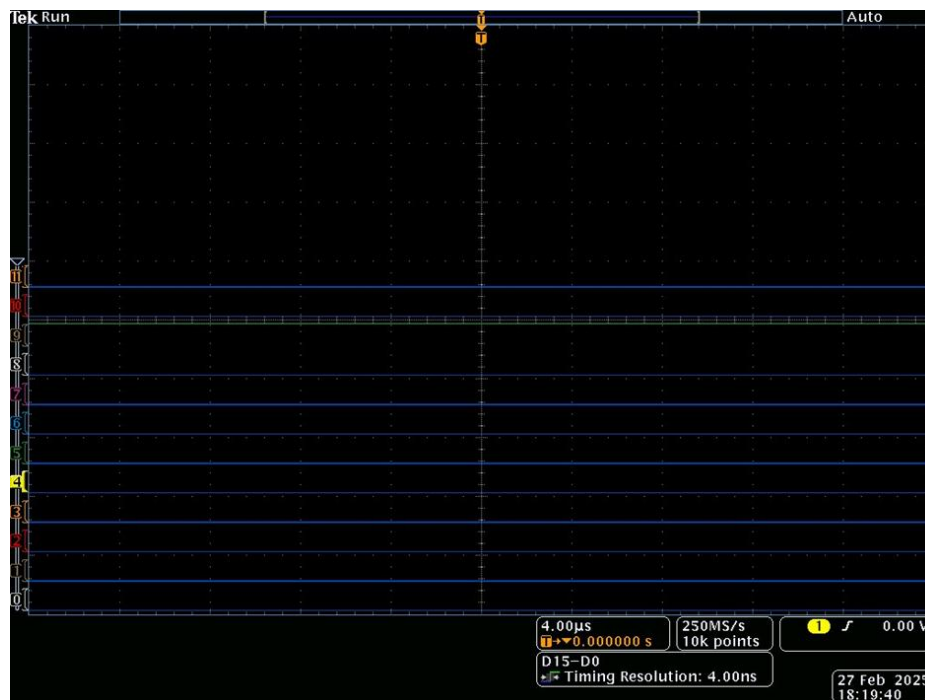


Figure 40 GPIO testing showing that a GPIO pin on the PCB can be driven HIGH and LOW.



Figure 41 GPIO testing showing that all 12 GPIO pins for a single channel are functional.

Serial – TCP bridge testing using SOCAT virtual serial ports, NECAT and SCREEN tools

The Serial-TCP bridge software in the bridge.py script and the UI in the index.html script was extensively tested using SOCAT and NECAT tools to setup virtual serial ports, establish TCP connections and send data between the virtual serial ports negating the need to interface with hardware and has been documented as a diagram in Figure 42. SOCAT and NETCAT are beneficial network testing tools because they support a variety of protocols including both TCP and UDP and various communications tunnels can be established including SCTP and HTTP. TTY and serial devices can also be connected over the network to be tested.

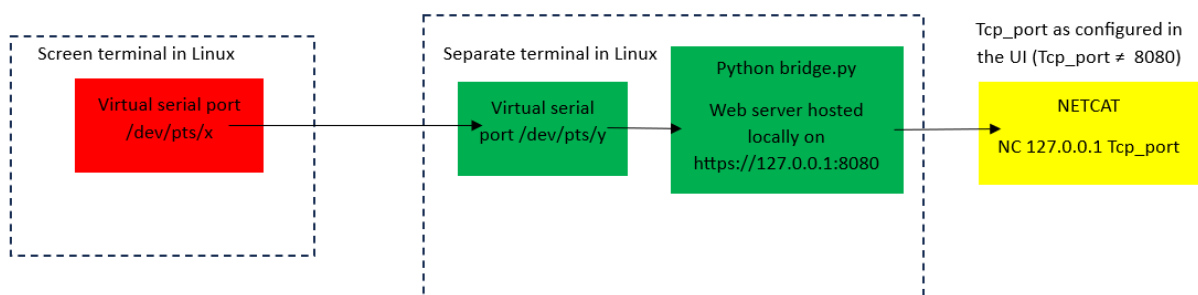


Figure 42 Testing using SOCAT virtual serial ports and NECAT tools.

The first terminal set up in Figure 43 shows the Serial to TCP bridge flask server running after the socat command has been run to set up the virtual serial port /dev/pts/x and connect it to the listening TCP_port where the number has been configured in the UI by the user. The second terminal shown in Figure 44 is running netcat and is the TCP client and sends/receives data over TCP simulating the remote device. The final terminal in Figure 45 monitors the serial port side and shows the serial data being transmitted.

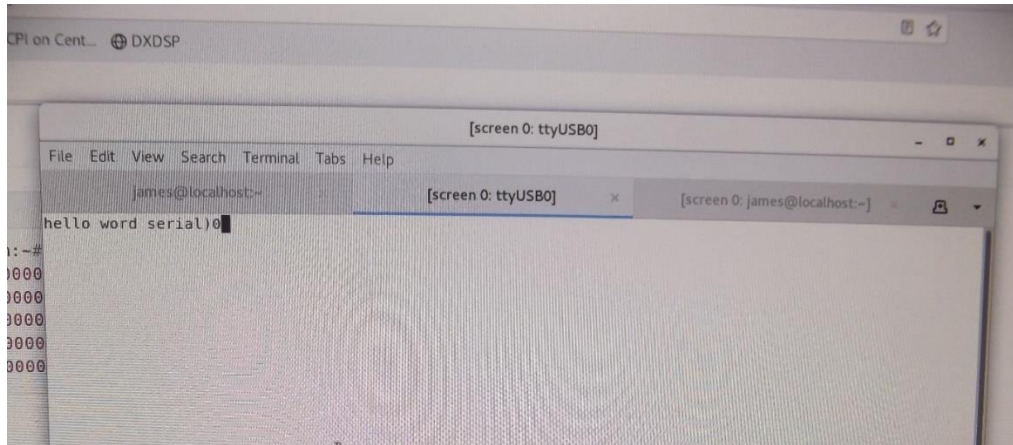


Figure 46 Screen terminal in Linux showing serial test data.

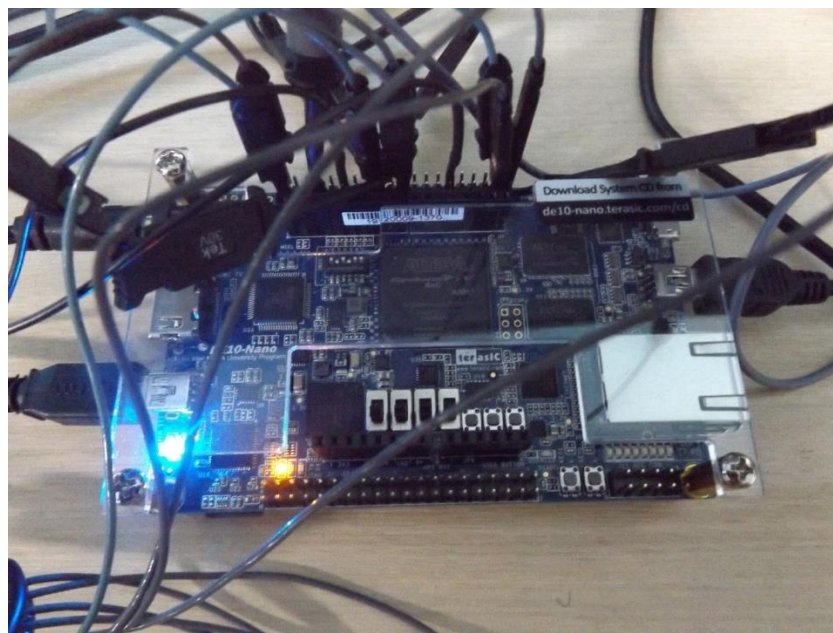


Figure 47 DE10 nano FPGA and logic analyser during serial port testing.

System testing

The full system was tested in the lab using the UI to configure a serial port with all parameters including the voltage switching which can only be tested using the hardware and then starting the bridge. The serial port was set to switch to 5.0v and this is successfully proven after the gpo_0(0) was successfully driven HIGH, and the voltage switched to 5.0v as had been configured. The evidence of this test is shown in Figure 48 with the server running on the left and an example configured port in the UI on the right. Figure 49 proves the GPIO line was correctly driven HIGH when the voltage switched to 5v. The second part in the system testing proved that the voltage could be correctly switched to 12v from the UI for one of the channel and evidence of this is shown in Figure 50 where the server is running on the left and an example configuration for a port switching to 12v is on the right. Figure 51 proves that gpo_0(4) is driven HIGH to 12v and both gpo_0(0) and gpo_(4) are showing on the oscilloscope screenshot as being successfully switched after configuring the voltage in the UI.

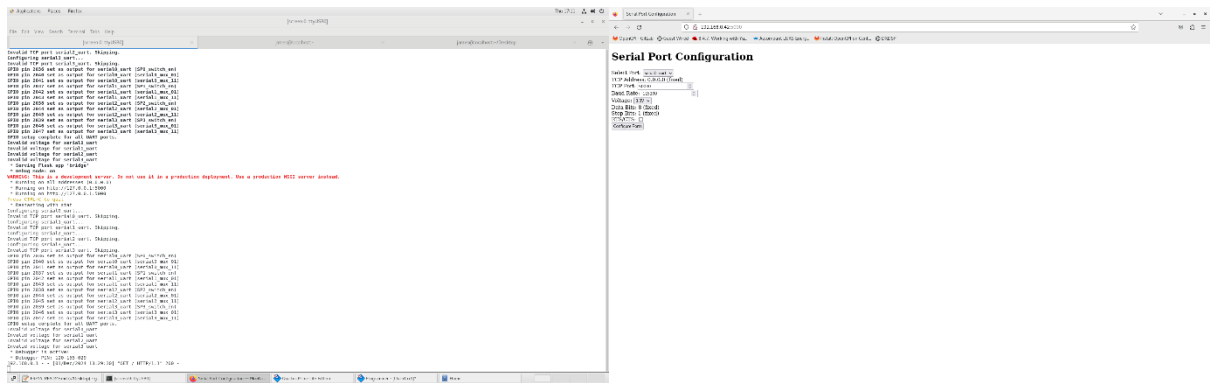


Figure 48 Full system test of serial0_uart channel where the voltage = 5.0v.

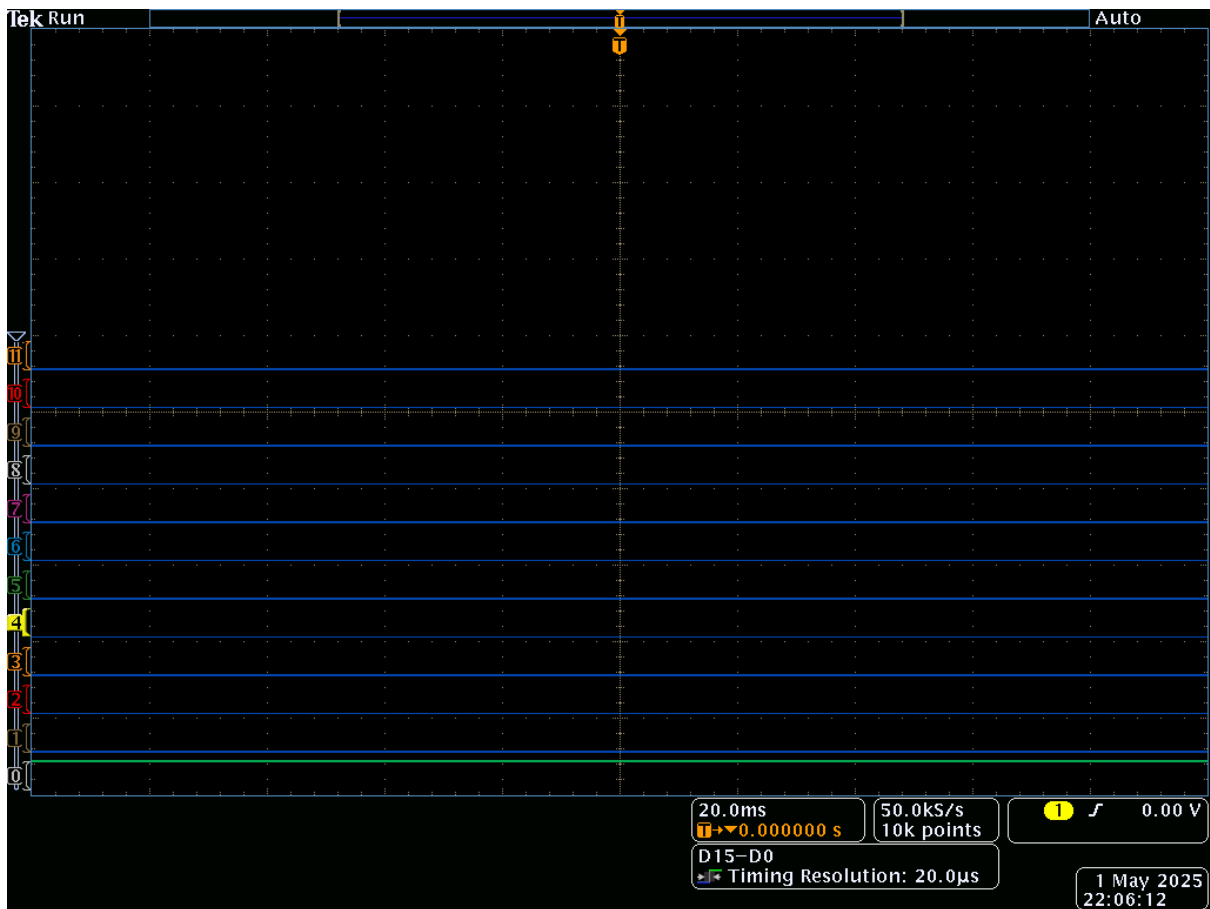


Figure 49 Full system testing showing GPIO pin 0 driven HIGH to 5.0v.

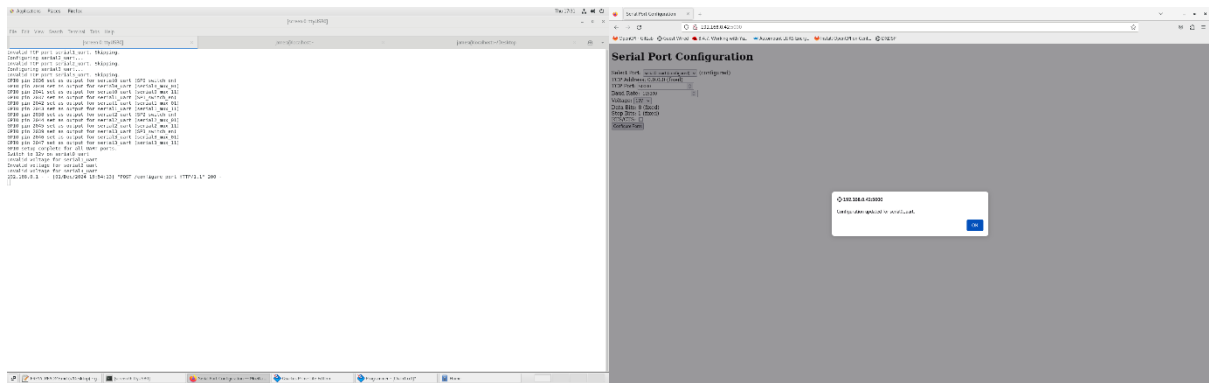


Figure 50 Full system test of serial0_uart channel where the voltage = 12.0v.

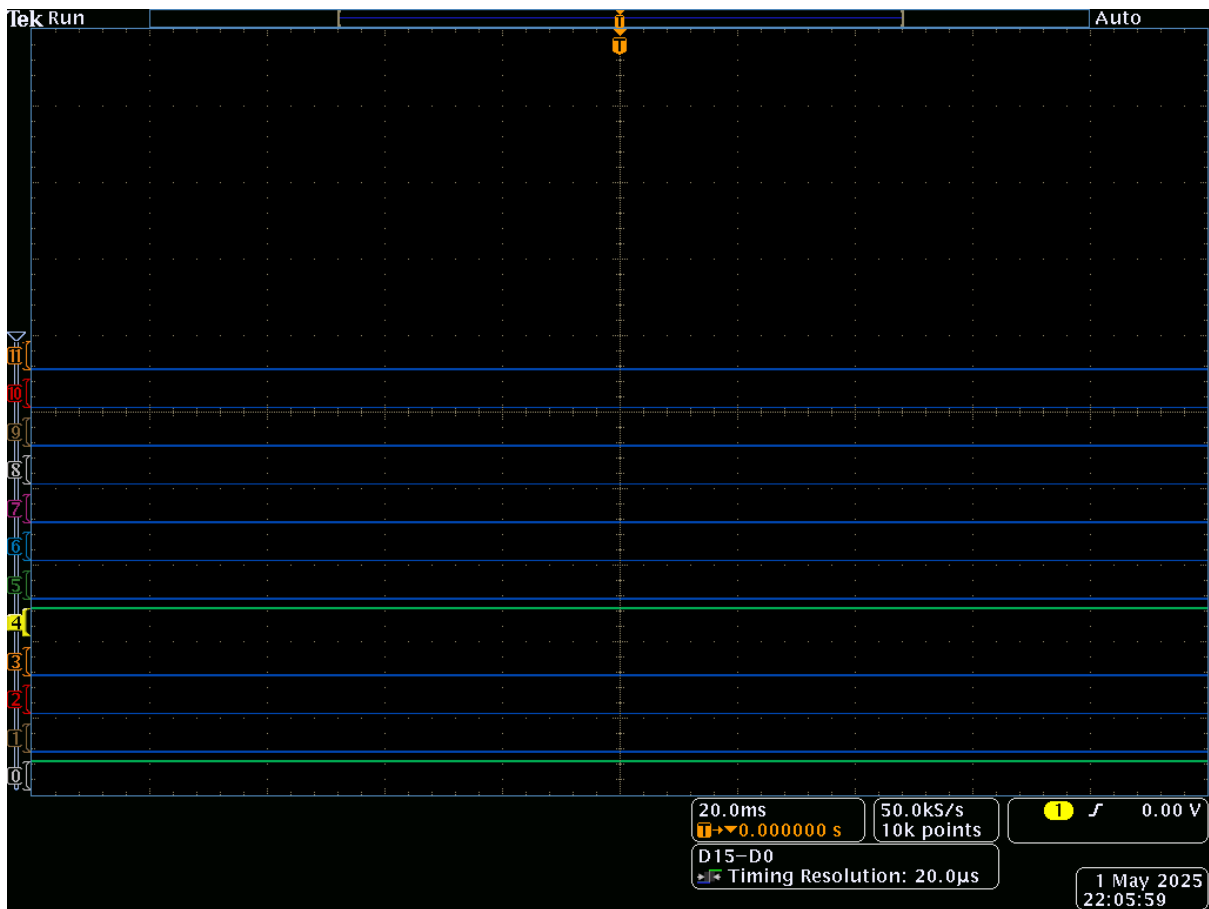


Figure 51 Full system testing showing GPIO pin 0 driven HIGH to 12.0v.

System provisioning guide

1. Save all required files – bridge.py, index.html and the SD card image (containing the de10_nano_clean_soc.sof and the de10_nano_clean_soc.dtb) onto the SD card to put into the DE10 nano.
2. Open a Linux terminal and run the command - sudo screen /dev/ttyUSB0 115200 (or a different USB baud rate if required but this unlikely).
3. Plug in the DE10 nano FPGA.
4. Stop the boot command by pressing space repeatedly.
5. Program the FPGA using the auto detect feature to find the correct device.
6. Type boot command into the uboot console.

7. Log in with root and root as username and password.
8. Run the command - `chmod +rw /sys/class/gpio/export`.
9. Run the command - `ip a add 192.168.0.42/24 dev end0` (this is only an example IP address)
10. Run the command - `ip link set dev end0 up`.
11. Check that the index.html script for the UI is located and saved in the templates folder on the DE10 nano SD card.
12. Run the Python script `python3 bridge.py` which should be on the SD card.

Conclusion

My final year project was successfully completed achieving the desired outcome of the wider project within my organisation because the embedded software and hardware device I have designed, built and tested can transmit or receive serial/TCP data and convert between the serial UART and TCP protocols between RS 232 and 0 – 3.3v/5v TTL by transmitting or receiving data in packets that are each 1024 bytes long. The hosted web server has the ability to concurrently transmit or receive data for four individually configured serial UART channels with configurable parameters including: baud rate of the serial device; TCP port number of the connected TCP port; enabling the use of RTS/CTS in the UART protocol; GPIO voltages at 3.3v, 5v or 12v and TCP address bound to all interfaces at 0.0.0.0, transmission of eight data bits at a time and inclusion of 1 stop bit for error checking are all fixed parameters. Over the course of the project, I learnt several new skills and developed my understanding and knowledge further in multiple areas which included developing an embedded Linux device tree for the UART channel interface using Debian specifically the Bookworm version as I had only previously done embedded Linux development using Yocto project.

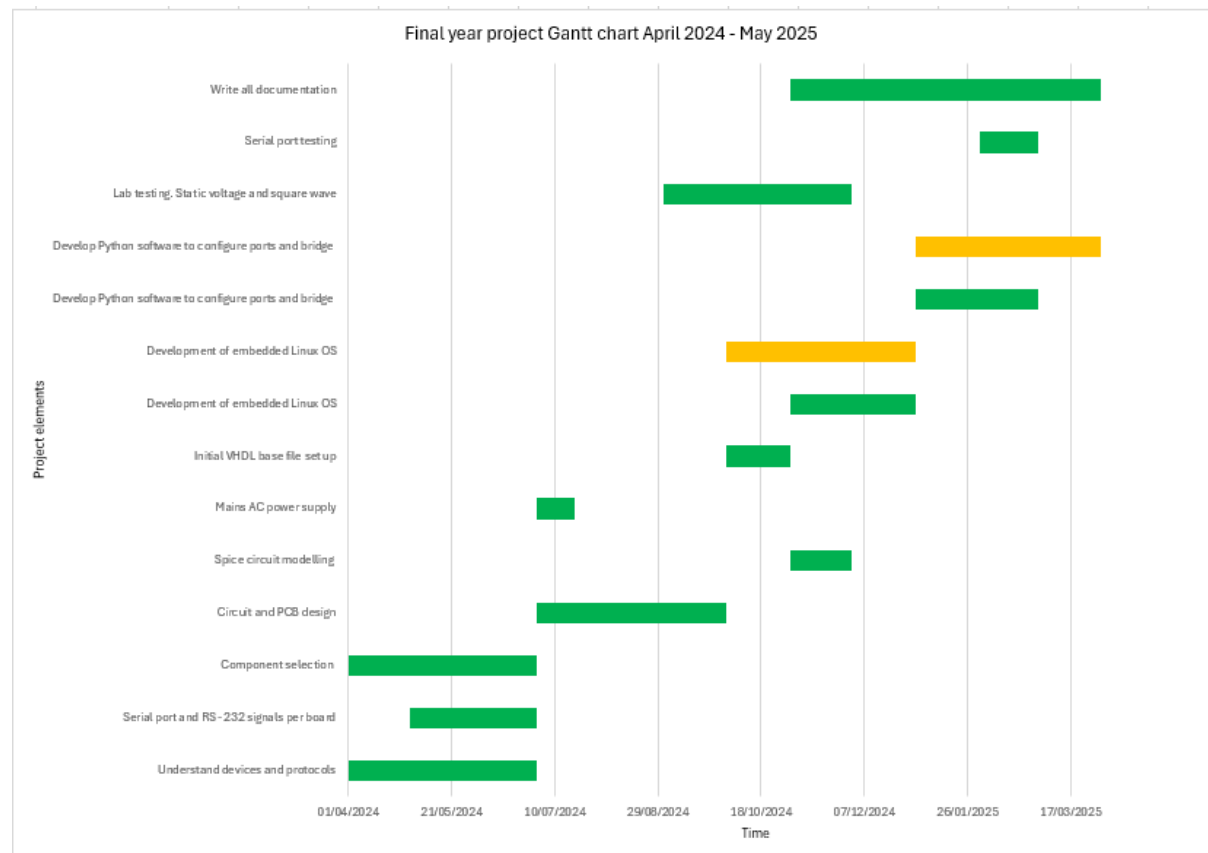


Figure 52 Final year project Gantt chart April 2024 – May 2025. Initial plan in GREEN and actual time for some tasks in ORANGE.

According to the Gantt chart in Figure x the development of the Python software for the Serial to TCP bridge and configuration took longer than was originally forecasted by approximately one month because it was to understand how to effectively implement and utilise the features of the Python 'select' library. It was challenging write the Python code to monitor the availability of the serial and TCP ports in a single thread alongside the file handles for the serial and TCP ports and then reading or writing the serial or TCP data to the appropriate port when it was available because I had never previously worked with threads or the 'select library'. It was difficult to implement it initially because I had to correctly store the Serial_port_file_handle, Tcp_port_file_handle and Tcp_client_file_handle in the global serial dictionary, shown in Figure x, and ensure that the 'select' library was looking for the correct file handle to ensure that it would check to see if the correct port was available.

Development on the DE10-Nano using Debian Bullseye was delayed due to UART driver incompatibility with the board's hardware in the older 5.x kernel. The device tree was configured correctly, but the Bullseye kernel lacked proper support for the Altera 16550-compatible UART, causing broken serial console output. Debugging was hindered due to the absence of a reliable UART interface. The issue was resolved by switching to Debian Bookworm, which includes a newer Linux kernel (6.x) with updated UART drivers and better support for the DE10-Nano's HPS peripherals. The kernel was rebuilt with custom device tree blobs matching the Cyclone V SoC hardware configuration. A new root filesystem based on Bookworm was generated using debootstrap and integrated into the SD card image.

Future improvements and development

There are several possibilities for future improvements and changes that could be made to this project for example, I have bound the tcp_address parameter to all interfaces with a fixed TCP address of 0.0.0.0, however in the future this could be modified if required so the user can configure the TCP address themselves.

Another potential improvement that could be made is there is an additional option to send and receive data in UDP form not just TCP data across the bridge between the TCP clients on the server side and the serial ports. UDP is generally quicker than TCP because there's no requirement to establish and maintain a connection because UDP is a 'send and forget' protocol which means data is sent without waiting for acknowledgements. An advantage of TCP is that the TCP protocol ensures that all data packets are received and reassembled in the correct order without corruption because the TCP protocol implements error checking with stop and parity bits. An advantage of UDP is that it has a lower latency speed so transmits serial data faster than TCP without the overhead of establishing and maintaining a connection so this would allow for the real time transfer of data.

The TCP protocol could be implemented initially to configure the serialX_uart channel parameters and the use of control messages for the acknowledgements of sending and receiving the serial or TCP data then switch to sending and receiving data using the UDP protocol when the data is being transmitted continuously.

References

1. [1] MAX232E+ IC voltage converter Analog Devices, *MAX232E+ IC Voltage Converter Datasheet*, MAX202E–MAX241E Series, Rev. 4, Aug. 2016. [Online]. Available: https://www.mouser.co.uk/datasheet/2/609/MAX202E_MAX241E-3127932.pdf
2. [2] Analogue multiplexer Analog Devices, *ADG5208F/ADG5209F Fault-Protected Analog Multiplexers Datasheet*, Rev. 0, Oct. 2016. [Online]. Available:

https://www.analog.com/media/en/technical-documentation/data-sheets/adg5208f_5209f.pdf

3. [3] Bidirectional logic shifter Texas Instruments, *TXS0108E 8-Bit Bi-Directional, Level-Shifting, Voltage Translator for Open-Drain and Push-Pull Applications*, Rev. L, Nov. 2024. [Online]. Available: <https://www.ti.com/lit/ds/symlink/txs0108e.pdf>
4. [4] DE10 NANO FPGA "DE10-Nano Development Board Documentation," *RocketBoards.org*, [Online]. Available: <https://www.rocketboards.org/foswiki/Documentation/DE10NanoDevelopmentBoard>. [Accessed: May 10, 2025].
5. [5] J. Gibbard, "Linux Debian on the DE10-Nano (2025)," *Gibbard.me*, [Online]. Available: https://www.gibbard.me/linux_debian_de10_2025/. [Accessed: May 10, 2025].
6. [6] "PySerial Documentation," *PythonHosted.org*, [Online]. Available: <https://pythonhosted.org/pyserial/>. [Accessed: May 10, 2025].
7. [7] "Flask Python Documentation," *Python-Adv-Web-Apps Documentation*, [Online]. Available: <https://python-adv-web-apps.readthedocs.io/en/latest/flask.html>. [Accessed: May 10, 2025].
8. [8] "Fundamentals of RS232 Serial Communications," *Analog.com*, [Online]. Available: <https://www.analog.com/en/resources/technical-articles/fundamentals-of-rs232-serial-communications.html>. [Accessed: May 10, 2025].
9. [9] "UART Communication," *Arduino Documentation*, [Online]. Available: <https://docs.arduino.cc/learn/communication/uart/>. [Accessed: May 10, 2025].
10. [10] "UART: A Hardware Communication Protocol," *Analog Devices*, [Online]. Available: <https://www.analog.com/en/resources/analog-dialogue/articles/uart-a-hardware-communication-protocol.html>. [Accessed: May 10, 2025].
11. [11] MEAN WELL, "RS-150-24 Switching Power Supply, 24 V dc, 6.5 A, 156 W, 1 Output, 176 – 264 V ac, 248 – 373 V dc," RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/switching-power-supplies/6210685>. [Accessed: 10-May-2025].
12. [12] MEAN WELL, "RD-50A Switching Power Supply, 5 V dc, 12 V dc, 2 A, 6 A, 54 W, Dual Output, 125 – 373 V dc, 88 – 264 V ac," RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/switching-power-supplies/6447067?gb=s>. [Accessed: 10-May-2025].
13. [13] TE Connectivity / Schaffner, "FN261-2-06 Power Entry Module with EMC Filter, 2 A, 250 V AC, Panel Mount, IEC Inlet," Mouser Electronics. [Online]. Available: <https://eu.mouser.com/ProductDetail/TE-Connectivity-Schaffner/FN261-2-06?qs=HsrPFI4P69wx%2FG6Em8j7hw%3D%3D>. [Accessed: 10-May-2025].
14. [14] TE Connectivity / Schaffner, "FN9260B-1-07 AC Power Entry Module with IEC Inlet Filter, 250 V AC, 1 A, Dual Fuse, Wire Leads," Mouser Electronics. [Online]. Available: <https://eu.mouser.com/ProductDetail/TE-Connectivity-Schaffner/FN9260B-1-07?qs=Li%252BoUPsLEnvX%2F%2Fuj%252BprNsA%3D%3D>. [Accessed: 10-May-2025].
15. [15] Texas Instruments, *TMUX7436F ±60 V Fault-Protected, Dual 2:1 Multiplexer With Latch-Up Immunity and 1.8 V Logic*, Datasheet, Rev. A, Nov. 2022. [Online]. Available: <https://www.ti.com/lit/ds/symlink/tmux7436f.pdf>. [Accessed: 10-May-2025].
16. [16] KEMET Electronics Corporation, *Surface Mount Multilayer Ceramic Chip Capacitors (SMD MLCCs): X7R Dielectric, 6.3 – 250 VDC (Automotive Grade)*, Datasheet, Rev. C1023_X7R_AUTO_SMD, Feb. 20, 2025. [Online]. Available: https://eu.mouser.com/datasheet/2/447/KEM_C1023_X7R_AUTO_SMD-3316698.pdf. [Accessed: 10-May-2025].

17. [17] CML Innovative Technologies, “1950X23X Series Red Panel Mount Indicator, 230 V AC, 8 mm Mounting Hole Size, IP67,” RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/panel-mount-indicators/1950324?gb=s>. [Accessed: 10-May-2025].
18. [18] RS PRO, “Green Panel Mount Indicator, 2 V DC, 8 mm Mounting Hole Size, Solder Tab Termination,” RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/panel-mount-indicators/0206132?gb=s>. [Accessed: 10-May-2025].
19. [19] RS PRO, “10-Way IDC Connector Plug for Through Hole Mount, 2-Row,” RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/idc-connectors/6257252?searchId=0ad61be6-c6c0-4a43-8c1c-ee3d9a8095d1&gb=s>. [Accessed: 10-May-2025].
20. [20] RS PRO, “10-Way IDC Connector Socket for Cable Mount, 2-Row,” RS Components, RS Online. [Online]. Available: <https://uk.rs-online.com/web/p/idc-connectors/6257369?searchId=7b97e802-7684-4ccb-97a1-e6b57faec22a&gb=s>. [Accessed: 10-May-2025].

Appendices

Component choices and information

Enough for 2.5 PCB or 12 channels initially Parts

	Option 1	Option 2
Power supply	150W 15V 10A Single Output Switching Power Supply - RS-150-24	Mean Well Switching Power Supply, RD-50A, 5 V dc, 12 V dc, 2 A, 6 A, 54W, Dual Output, 125 → 373 V dc, 88 RS Stock Number - 644-7067 Mfr. Part Number - RD-50A
Link to purchase	Mean Well Switching Power Supply, RS-150-15, 15V dc, 10A, 150W, 1 Output, 176 → 264 V ac, 248 → 373 V dc, 1 RS (rs-online.com)	https://uk.rs-online.com/web/p/switching-power-supplies/6447067
Price	£33.13	£19.49
Quantity	1	1
Price * Quantity	£33.13	£19.49
Link to datasheet	未命名 -1 (rs-online.com)	n/a

Figure 53 Comparison of mains power supply. Option 1 [11] Option 2 [12]

	Option 1	Option 2
IEC embedded EMC filter	IEC embedded EMC filter Mouser number - 631-FN261-2/06. Mfr Part number FN261-2-06	IEC embedded EMC filter Mouser number - 631-FN9260B-1-07 Mfr. Part number - FN9260B-1-07
Link to purchase	https://www.mouser.co.uk/ProductDetail/Schaffner/FN261-2-06?qs=HsrPF14P69wx%2FG6Em8j7hw%3D%3D	https://www.mouser.co.uk/ProductDetail/Schaffner/FN9260B-1-07?qs=Li%252BoUPsLEnvX%2F%2Fuj%252BprNsA%3D%3D
Price	£13.52	£15.61
Quantity	1	1
Price * Quantity	£13.52	£15.61
Link to datasheet	https://www.mouser.co.uk/datasheet/2/355/Schaffner_datasheet_FN9260_FN261-1730572.pdf	https://www.mouser.co.uk/datasheet/2/355/Schaffner_datasheet_FN9260_FN261-1730572.pdf

Figure 54 Comparison of IEC embedded EMC filter. Option 1 [13] Option 2 [14]

	Option 1	Option 2
Analogue multiplexer	Analogue multiplexer Mouser number - 584-ADG5209FBRUZ Mfr. number - ADG5209FBRUZ	Analogue multiplexer Mouser number - 595-TMUX7436FPWR Mfr. number - TMUX7436FPWR
Link to purchase	https://www.mouser.co.uk/ProductDetail/Analog-Devices/ADG5209FBRUZ?qs=4BLsKd%2FI MYAPq9eVP6Wx0g%3D%3D	https://www.mouser.co.uk/ProductDetail/Texas-Instruments/TMUX7436FPWR?qs=IPgv5n7u5Qa5KhfUbki hcw%3D%3D
Price	£9.59	£8.03
Quantity	24	32
Price * Quantity	£230.16	£256.96
Link to datasheet	https://www.mouser.co.uk/datasheet/2/609/adg5208f_5209f-3120832.pdf	https://www.ti.com/lit/ds/symlink/tmux7436f.pdf?HQS=dis-mous-null-mouser-mode-dsf-pf-null-www&ts=1695037724235&ref_url=https%253A%252F%252Fwww.mouser.es%252F

Figure 55 Comparison of analogue multiplexer. Option 1 [2] Option 2 [15]

	Option 1
MAX232x integrated circuit chip	RS-232 Interface IC +/-15kV ESD-Protected, +5V RS-232 Transceivers MAX232x IC chip Mouser number - 700-MAX232ECWET Mfr. number MAX232ECWE+T
Link to purchase	https://www.mouser.co.uk/ProductDetail/Analog-Devices-Maxim-Integrated/MAX232ECWE%2bT?qs=CDqwynd4ZNqsGylgy%252BI5xA%3D%3D
Price	£5.12
Quantity	12
Price * Quantity	£61.44
Link to datasheet	https://www.mouser.co.uk/datasheet/2/609/MAX202E_MAX241E-3127932.pdf

Figure 56 MAX232x IC voltage converter information. [1]

	Option 1
Mains Power LED 230v	CML Innovative Technologies 1950X23X Series Red Panel Mount Indicator, 230V ac, 8mm Mounting Hole Size, IP67 RS Stock number - 195-0324. Mfr. Part number - 19500230
Link to purchase	https://uk.rs-online.com/web/p/panel-mount-indicators/1950324?gb=s
Price	£6.87
Quantity	1
Price * Quantity	£6.87
Link to datasheet	https://docs.rs-online.com/bc6c/A700000006548090.pdf

Figure 57 Mains power LED information. [17]

	Option 1
FPGA Power LED 5.0v	RS PRO Green Panel Mount Indicator, 2V dc, 8mm Mounting Hole Size, Solder Tab Termination RS Stock number - 206-132.
Link to purchase	https://uk.rs-online.com/web/p/panel-mount-indicators/0206132?gb=s
Price	£3.44
Quantity	1
Price * Quantity	£3.44
Link to datasheet	https://docs.rs-online.com/8764/0900766b815afeef.pdf

Figure 58 FPGA power LED information. [18]

	Option 1
<u>10 way</u> IDC PCB mount	RS PRO 10-Way IDC Connector Plug for Through Hole Mount, 2-Row RS Stock Number - 625-7252
Link to purchase	https://uk.rs-online.com/web/p/idc-connectors/6257252?searchId=edafdf5-d638-42e6-b4a8-75be2e866b1f&gb=s
Price	£3.25
Quantity	12
Price * Quantity	£39
Link to datasheet	https://docs.rs-online.com/9f3d/A700000008919302.pdf

Figure 59 10-way PCB mount. [19]

	Option 1
10 way IDC cable mount	RS PRO 10-Way IDC Connector Socket for Cable Mount, 2-Row RS Stock Number - 625-7369
Link to purchase	https://uk.rs-online.com/web/p/idc-connectors/6257369?searchId=28336a10-caf2-4832-a0a6-41ab70526fc3&gb=s
Price	£1.53
Quantity	12
Price * Quantity	£18.36
Link to datasheet	https://docs.rs-online.com/390a/A700000007925664.pdf

Figure 60 10-way cable mount. [20]

	Option 1
0.1uF ceramic capacitor	Multilayer Ceramic Capacitors MLCC - SMD/SMT 100V 100000pF 10% X7R 1608 AEC-Q200 Mouser Number - 80-C0603C104K1RAUTO Mfr. Number - C0603C104K1RACAUTO
Link to purchase	https://www.mouser.co.uk/ProductDetail/KEMET/C0603C104K1RAUTO?qs=3Ra4i%252BhyCGTQ5uPNXtfZA%3D%3D
Price	£0.24
Quantity	140
Price * Quantity	£33.60
Link to datasheet	https://www.mouser.co.uk/datasheet/2/447/KEM_C1023_X7R_AUTO_SMD-3316698.pdf

Figure 61 0.1uF ceramic capacitor. [16]

	Option 1
Ribbon cables	RS PRO Flat Ribbon Cable, 10-Way, 1.27mm Pitch, 30m Length RS Stock Number - 360-093
Link to purchase	https://uk.rs-online.com/web/p/ribbon-cable/0360093?searchId=1d4628dc-842d-4c89-901e-dd1c82deedfe&gb=s
Price	£29.14
Quantity	5 * 30m
Price * Quantity	£145.70
Link to datasheet	none

Python bridge.py Serial to TCP bridge and configuration software

```
from flask import Flask, jsonify, request, render_template
from flask import url_for
import threading, time, serial, socket, select
import gpio as GPIO
```

```
app = Flask(__name__)
```

```

# Global variable for the monitor_thread function
monitor_thread = None
stop_event = threading.Event() # Defined stop_event for thread control

# Global lists for serial and TCP port file handles

# Global dictionary for serial ports, now includes port settings and file
handles
serial_ports = {
    "serial0_uart": {
        "baud_rate": 0, # Initial baud rate set to 0, configurable by the
user
        "tcp_port": None, # Initial tcp port number set to 0,
configurable by the user
        "rts_cts": False, # By default, RTS/CTS is not enabled
        "serial_device": "/dev/ttyAL0",
        "data_bits": 8, # Data bits fixed at 8
        "stop_bits": 1, # Stop bits fixed at 1
        "voltage": 0, # Initial voltage set to 0, configurable by the
user
        "tcp_address": "0.0.0.0",
        "serial_port_file_handle": None,
        "tcp_port_file_handle": None,
        "tcp_client_file_handle": None,
        "configured": False # Flag to indicate if the port is configured
    },
    "serial1_uart": {
        "baud_rate": 0,
        "tcp_port": None,
        "rts_cts": False,
        "serial_device": "/dev/ttyAL1",
        "data_bits": 8,
        "stop_bits": 1,
        "voltage": 0,
        "tcp_address": "0.0.0.0",
        "serial_port_file_handle": None,
        "tcp_port_file_handle": None,
        "tcp_client_file_handle": None,
        "configured": False
    },
    "serial2_uart": {
        "baud_rate": 0,
        "tcp_port": None,
        "rts_cts": False,
        "serial_device": "/dev/ttyAL2",
        "data_bits": 8,
        "stop_bits": 1,
        "voltage": 0,
        "tcp_address": "0.0.0.0",
        "serial_port_file_handle": None,
        "tcp_port_file_handle": None,
        "tcp_client_file_handle": None,
        "configured": False
    },
    "serial3_uart": {
        "baud_rate": 0,
        "tcp_port": None,
        "rts_cts": False,
        "serial_device": "/dev/ttyAL3",
        "data_bits": 8,
        "stop_bits": 1,

```

```

        "voltage": 0,
        "tcp_address": "0.0.0.0",
        "serial_port_file_handle": None,
        "tcp_port_file_handle": None,
        "tcp_client_file_handle": None,
        "configured": False
    }
}

# GPIO configuration for ports
gpio_offset = 2036
gpio_config = {
    "serial0_uart": {
        "SP0_switch_en": gpio_offset,
        "serial0_mux_01": gpio_offset + 4,
        "serial0_mux_11": gpio_offset + 5
    },
    "serial1_uart": {
        "SP1_switch_en": gpio_offset + 1,
        "serial1_mux_01": gpio_offset + 6,
        "serial1_mux_11": gpio_offset + 7
    },
    "serial2_uart": {
        "SP2_switch_en": gpio_offset + 2,
        "serial2_mux_01": gpio_offset + 8,
        "serial2_mux_11": gpio_offset + 9
    },
    "serial3_uart": {
        "SP3_switch_en": gpio_offset + 3,
        "serial3_mux_01": gpio_offset + 10,
        "serial3_mux_11": gpio_offset + 11
    }
}

# GPIO setup function
def setup_gpio_for_ports():
    try:
        for port in gpio_config:
            for pin_name in gpio_config[port]:
                pin = gpio_config[port][pin_name]
                GPIO.setup(pin, GPIO.OUT)
                print(f"GPIO pin {pin} set as output for {port}
({pin_name})")
            print("GPIO setup complete for all UART ports.")
        except Exception as e:
            print(f"GPIO setup error: {e}")

    if True:
        for port in gpio_config:
            # Get lines for the necessary GPIO pins based on the
configuration
            sp_switch_en = gpio_config[port][f"SP{port[6]}_switch_en"]
            serial_mux_01 = gpio_config[port][f"serial{port[6]}_mux_01"]
            serial_mux_11 = gpio_config[port][f"serial{port[6]}_mux_11"]

            voltage = serial_ports[port]["voltage"]

            # Set the values based on the combination
            if voltage == '3.3V':
                GPIO.output(sp_switch_en, GPIO.LOW)
                GPIO.output(serial_mux_01, GPIO.LOW)

```

```

        GPIO.output(serial_mux_11, GPIO.LOW)
        print(f"Switch to 3.3v on {port}")
    elif voltage == '5.0V':
        GPIO.output(sp_switch_en, GPIO.HIGH)
        GPIO.output(serial_mux_01, GPIO.HIGH)
        GPIO.output(serial_mux_11, GPIO.LOW)
        print(f"Switch to 5.0v on {port}")
    elif voltage == '12.0V':
        GPIO.output(sp_switch_en, GPIO.HIGH)
        GPIO.output(serial_mux_01, GPIO.LOW)
        GPIO.output(serial_mux_11, GPIO.HIGH)
        print(f"Switch to 12v on {port}")
    else:
        print(f"Invalid voltage for {port}")

# except Exception as e:

#     print(f"Error configuring GPIOs for {port}: {e}")

def configure_ports():

    # Open Serial Ports and TCP Ports
    for port, config in serial_ports.items():
        print(f"Configuring {port}...")

        # Open Serial Port if configured
        if config["baud_rate"] != 0 and config["serial_device"]:
            try:
                serial_port = serial.Serial(config["serial_device"],
config["baud_rate"], timeout=1)
                config["serial_port_file_handle"] = serial_port
                print(f"Opened serial port {config['serial_device']} at
{config['baud_rate']} baud.")
            except Exception as e:
                print(f"Error opening serial port
{config['serial_device']}: {e}")

            # Handle TCP Port configuration
            tcp_port = config.get("tcp_port")
            try:
                # Ensure tcp_port is an integer
                if tcp_port:
                    tcp_port = int(tcp_port) # Ensure the tcp_port is cast to
an integer

                    if isinstance(tcp_port, int) and 1 <= tcp_port <= 65535:
                        # Only open TCP port if it's a valid port (non-zero)
                        if tcp_port != 0:
                            tcp_socket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
                            tcp_socket.setsockopt(socket.SOL_SOCKET,
socket.SO_REUSEADDR, 1)
                            tcp_socket.bind((config["tcp_address"], tcp_port))
                            tcp_socket.listen(1) # Listen for one connection
at a time

                            config["tcp_port_file_handle"] = tcp_socket
                            print(f"Opened TCP socket on
{config['tcp_address']}:{tcp_port}")
                        else:
                            print(f"Invalid TCP port {port}. Skipping.")
                    else:

```

```

        print(f"Invalid TCP port {port}. It must be an integer
between 1 and 65535. Skipping.")
    else:
        print(f"Invalid TCP port {port}. Skipping.")

    except ValueError:
        print(f"Error: TCP port {port} should be an integer, but got
{type(tcp_port)}")
    except Exception as e:
        print(f"Error opening TCP socket on
{config['tcp_address']}:{tcp_port}: {e}")
        threading.Thread(target=handle_ports, daemon=True).start() # Start
communication handling in the background
        setup_gpio_for_ports()

# Function to close all file handles (both serial and TCP)
def close_all_file_handles():
    try:
        for port, config in serial_ports.items():
            if config["serial_port_file_handle"]:
                config["serial_port_file_handle"].close()
            if config["tcp_port_file_handle"]:
                config["tcp_port_file_handle"].close()
    except Exception as e:
        print(f"Error closing serial and tcp ports")

# Function to handle communication between serial and TCP ports
def handle_ports():
    try:
        # Initialize a list to track all active ports
        all_ports = []

        # Add all serial ports that are open
        for port, config in serial_ports.items():
            serial_port = config["serial_port_file_handle"]
            if serial_port and serial_port.is_open: # Check if serial port
is initialized and open
                all_ports.append(serial_port)

        # Add all TCP sockets that are open
        for port, config in serial_ports.items():
            tcp_socket = config["tcp_port_file_handle"]
            if tcp_socket and tcp_socket.fileno() != -1: # Check if TCP
socket is valid
                all_ports.append(tcp_socket)

        # Continuously monitor ready sockets for communication
        while True:
            ready_sockets, _, _ = select.select(all_ports, [], [], 1) #
Monitor all active ports
            for s in ready_sockets:
                for port, config in serial_ports.items():
                    # Check if the source is a serial port file handle
                    if s == config["serial_port_file_handle"]:
                        if config["serial_port_file_handle"] and
config["serial_port_file_handle"].is_open:
                            data = s.read() # Read data from serial port
                            if config["tcp_client_file_handle"] is None:
                                # Handle case when TCP client is not
initialized
                                print(f"TCP client for {port} is not set.")

```

```

        else:
            # Send data over the TCP connection

config["tcp_client_file_handle"].sendall(data)
            # Debugging output
            print(f"[{port}] Sent to TCP: {data!r}")
        else:
            print(f"Serial port for {port} is not available
or closed.")

            # Check if the source is a TCP port file handle
            elif s == config["tcp_port_file_handle"]:
                if config["tcp_port_file_handle"]:
                    conn, addr = s.accept() # Accept incoming TCP
connection

                    config["tcp_client_file_handle"] = conn
                    all_ports.append(conn) # Track all active
connections

                    print(f"Connection accepted from {addr} for
{port}")
                else:
                    print(f"TCP port for {port} is not set.")

            # Handle incoming TCP data
            elif s == config["tcp_client_file_handle"]:
                if config["tcp_client_file_handle"]:
                    data = s.recv(1024) # Receive data from TCP
client

                    if config["serial_port_file_handle"] and
config["serial_port_file_handle"].is_open:
                        # Write received TCP data to the serial
port

                        config["serial_port_file_handle"].write(data)
                        print(f"[{port}] Received from TCP and
wrote to Serial: {data!r}")
                    else:
                        print(f"Serial port for {port} is not
available or closed.")
                else:
                    print(f"TCP client for {port} is not set.")
            except Exception as e:
                print(f"Error in handle_ports: {e}")

# Function to handle TCP communication in a separate thread
def handle_tcp_communication(conn, serial_port, tcp_socket, port):
    try:
        while True:
            data = conn.recv(1024) # Read data from TCP connection
            if data:
                # Write the received TCP data to the corresponding serial
port

                try:
                    if serial_port and serial_port.is_open:
                        serial_port.write(data) # Send data to serial port
                        print(f"Data sent to serial port
{serial_port.name}: {data}")
                    else:
                        print(f"Serial port {serial_port.name} is not
open!")
                except Exception as e:

```

```

        print(f"Error writing data to serial port
{serial_port.name}: {e}")
    else:
        break # No more data, break the loop
except Exception as e:
    print(f"Error handling TCP communication on port {port}: {e}")
finally:
    conn.close() # Close the TCP connection
    print(f"Connection closed for TCP port {port}")

# Function to reset and reconfigure the ports based on new parameters
def reconfigure_port(port, baud_rate, tcp_port):
    # Close and discard current connections and file handles
    if serial_ports[port]["serial_port_file_handle"]:
        serial_ports[port]["serial_port_file_handle"].close()
        print(f"Closed current serial port {port}")

    if serial_ports[port]["tcp_port_file_handle"]:
        serial_ports[port]["tcp_port_file_handle"].close()
        print(f"Closed current TCP socket for {port}")

    # Update the configuration
    serial_ports[port]["baud_rate"] = baud_rate
    serial_ports[port]["tcp_port"] = tcp_port
    serial_ports[port]["configured"] = False

    # Reconfigure the ports with the new settings
    configure_ports()

# Flask route to configure serial port settings
@app.route('/configure_port', methods=['POST'])

def configure_port():
    data = request.get_json()
    port = data.get('port')
    baud_rate = data.get('baud_rate')
    tcp_port_number = data.get('tcp_port')
    voltage = data.get('voltage')

    # Ensure all data is present and TCP port is an integer
    if not port or baud_rate is None or tcp_port_number is None:
        return jsonify({"error": "Port, baud_rate, and tcp_port are
required"}), 400

    try:
        tcp_port_number = int(tcp_port_number) # Ensure tcp_port_number is
an integer
    except ValueError:
        return jsonify({"error": "tcp_port should be an integer"}), 400

    if port in serial_ports:
        serial_ports[port]["baud_rate"] = baud_rate
        serial_ports[port]["tcp_port"] = tcp_port_number
        serial_ports[port]["configured"] = True
        serial_ports[port]["voltage"] = voltage

        # Close all open file handles
        close_all_file_handles()

        # Reconfigure the ports
        configure_ports()

```

```

        return jsonify({"message": f"Configuration updated for {port}."}),
200
    else:
        return jsonify({"error": "Invalid port name"}), 400

@app.route('/')
def index():
    return render_template('index.html')

# Start the Flask server and background communication thread
if __name__ == "__main__":
    configure_ports() # Initial port configuration

try:
    app.run(host='0.0.0.0', port=5000, debug=True)
except Exception as e:
    print(f"Error starting Flask server: {e}")

```

Embedded Linux device tree code

```

fpga_uart0: fpga_uart0@C0000600
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000600 0x200>;
    interrupts = <72>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart1: fpga_uart1@C0000400
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000400 0x200>;
    interrupts = <73>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart2: fpga_uart2@C0000200
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000200 0x200>;
    interrupts = <74>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_uart3: fpga_uart3@C0000000
{
    compatible = "altr,16550-FIFO128";
    reg = <0xC0000000 0x200>;
    interrupts = <75>;
    clock-frequency = <50000000>;
    reg-shift = <2>;
    fifo-size = <128>;
};

fpga_pio0: fpga_pio0@C0000800
{
    compatible = "altr,pio-1.0";
    reg = <0xC0000800 0x20>;
    altr,ngpio=<12>;
    #gpio-cells=<2>;
    gpio-controller;
};

```

UART channel VHDL code

```

-- UART 1
signal SP1_switch_en      : out std_logic;
signal serial1_mux_01     : out std_logic;
signal serial1_mux_11     : out std_logic;
signal serial1_TX_3_3V    : out std_logic;
signal serial1_RTS_3_3V   : out std_logic;
signal serial1_CTS_3_3V0 : in  std_logic;
signal serial1_RX_3_3V0   : in  std_logic;
signal serial1_CTS_3_3V1 : in  std_logic;
signal serial1_RX_3_3V1   : in  std_logic;

-- UART 2
signal SP2_switch_en      : out std_logic;
signal serial2_mux_01     : out std_logic;
signal serial2_mux_11     : out std_logic;
signal serial2_TX_3_3V    : out std_logic;
signal serial2_RTS_3_3V   : out std_logic;
signal serial2_CTS_3_3V0 : in  std_logic;
signal serial2_RX_3_3V0   : in  std_logic;
signal serial2_CTS_3_3V1 : in  std_logic;
signal serial2_RX_3_3V1   : in  std_logic;

-- UART 3
signal SP3_switch_en      : out std_logic;
signal serial3_mux_01     : out std_logic;
signal serial3_mux_11     : out std_logic;
signal serial3_TX_3_3V    : out std_logic;
signal serial3_RTS_3_3V   : out std_logic;
signal serial3_CTS_3_3V0 : in  std_logic;
signal serial3_RX_3_3V0   : in  std_logic;
signal serial3_CTS_3_3V1 : in  std_logic;
signal serial3_RX_3_3V1   : in  std_logic
);
end de10_nano_clean;

architecture behavioral of de10_nano_clean is

    signal reset : std_logic;
    signal gpo_0 : std_logic_vector(11 downto 0);
    signal serial0_RX_combined : std_logic;
    signal serial0_CTS_combined : std_logic;
    signal serial1_RX_combined : std_logic;
    signal serial1_CTS_combined : std_logic;
    signal serial2_RX_combined : std_logic;
    signal serial2_CTS_combined : std_logic;
    signal serial3_RX_combined : std_logic;
    signal serial3_CTS_combined : std_logic;

    hps_io_hps_io_spim1_inst_CLK : out std_logic;
    hps_io_hps_io_spim1_inst_MOSI : out std_logic;
    hps_io_hps_io_spim1_inst_MISO : in  std_logic := 'X';
    hps_io_hps_io_spim1_inst_SS0 : out std_logic;
    hps_io_hps_io_uart0_inst_RX : in  std_logic := 'X';
    hps_io_hps_io_uart0_inst_TX : out std_logic;
    hps_io_hps_io_i2c0_inst_SDA : inout std_logic := 'X';
    hps_io_hps_io_i2c0_inst_SCL : inout std_logic := 'X';
    hps_io_hps_io_i2c1_inst_SDA : inout std_logic := 'X';
    hps_io_hps_io_i2c1_inst_SCL : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO09 : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO35 : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO40 : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO53 : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO54 : inout std_logic := 'X';
    hps_io_hps_io_gpio_inst_GPIO61 : inout std_logic := 'X';
    hps_0_h2f_reset_reset_n : out std_logic;
    uart_0_rxd : in  std_logic := 'X';
    uart_0_txd : out std_logic;
    uart_0_cts_n : in  std_logic := 'X';
    uart_0_rts_n : out std_logic;
    uart_1_rxd : in  std_logic := 'X';
    uart_1_txd : out std_logic;
    uart_1_cts_n : in  std_logic := 'X';
    uart_1_rts_n : out std_logic;
    uart_2_rxd : in  std_logic := 'X';
    uart_2_txd : out std_logic;
    uart_2_cts_n : in  std_logic := 'X';
    uart_2_rts_n : out std_logic;
    uart_3_rxd : in  std_logic := 'X';
    uart_3_txd : out std_logic;
    uart_3_cts_n : in  std_logic := 'X';
    uart_3_rts_n : out std_logic;
    gpo_0_export : out std_logic_vector(11 downto 0)

    -- reset_n
    -- rxd
    -- txd
    -- cts_n
    -- rts_n
    -- rxd
    -- txd
    -- cts_n
    -- rts_n
    -- rxd
    -- txd
    -- cts_n
    -- rts_n
    -- rxd
    -- txd
    -- cts_n
    -- rts_n
);
end component de10_nano_clean_soc;

```

```

hps_0_h2f_reset_reset_n    => reset,                                -- hps_0_h2f_reset.reset_n
uart_0_rxd                  => serial0_RX_combined, -- uart_0.rxd
uart_0_txd                  => serial0_TX_3_3V,      --.txd
uart_0_cts_n                => serial0_CTS_combined, --.cts_n
uart_0_rts_n               => serial0_RTS_3_3V,      --.rts_n
uart_1_rxd                  => serial1_RX_combined, -- uart_1.rxd
uart_1_txd                  => serial1_TX_3_3V,      --.txd
uart_1_cts_n               => serial1_CTS_combined, --.cts_n
uart_1_rts_n               => serial1_RTS_3_3V,      --.rts_n
uart_2_rxd                  => serial2_RX_combined, -- uart_2.rxd
uart_2_txd                  => serial2_TX_3_3V,      --.txd
uart_2_cts_n               => serial2_CTS_combined, --.cts_n
uart_2_rts_n               => serial2_RTS_3_3V,      --.rts_n
uart_3_rxd                  => serial3_RX_combined, -- uart_3.rxd
uart_3_txd                  => serial3_TX_3_3V,      --.txd
uart_3_cts_n               => serial3_CTS_combined, --.cts_n
uart_3_rts_n               => serial3_RTS_3_3V,      --.rts_n
gpo_0_export               => gpo_0                  -- gpo_0.export
);

-- Switch_en, MUX_01 and MUX_11 GPOs lines
SP0_switch_en <= gpo_0(0); -- SP0_switch_en GPO line
SP1_switch_en <= gpo_0(1); -- SP1_switch_en GPO line
SP2_switch_en <= gpo_0(2); -- SP2_switch_en GPO line
SP3_switch_en <= gpo_0(3); -- SP3_switch_en GPO line
serial0_mux_01 <= gpo_0(4); -- serial0_MUX_01 GPO line
serial0_mux_11 <= gpo_0(5); -- serial0_MUX_11 GPO line
serial1_mux_01 <= gpo_0(6); -- serial1_MUX_01 GPO line
serial1_mux_11 <= gpo_0(7); -- serial1_MUX_11 GPO line
serial2_mux_01 <= gpo_0(8); -- serial2_MUX_01 GPO line
serial2_mux_11 <= gpo_0(9); -- serial2_MUX_11 GPO line
serial3_mux_01 <= gpo_0(10); -- serial3_MUX_01 GPO line
serial3_mux_11 <= gpo_0(11); -- serial3_MUX_11 GPO line
serial0_RX_combined <= serial0_RX_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_RX_3_3V1;
serial0_CTS_combined <= serial0_CTS_3_3V0 when (gpo_0(0) = '0' and gpo_0(4) = '0' and gpo_0(5) = '0') else serial0_CTS_3_3V1;
serial1_RX_combined <= serial1_RX_3_3V0 when (gpo_0(1) = '0' and gpo_0(6) = '0' and gpo_0(7) = '0') else serial1_RX_3_3V1;
serial1_CTS_combined <= serial1_CTS_3_3V0 when (gpo_0(1) = '0' and gpo_0(6) = '0' and gpo_0(7) = '0') else serial1_CTS_3_3V1;
serial2_RX_combined <= serial2_RX_3_3V0 when (gpo_0(2) = '0' and gpo_0(8) = '0' and gpo_0(9) = '0') else serial2_RX_3_3V1;
serial2_CTS_combined <= serial2_CTS_3_3V0 when (gpo_0(2) = '0' and gpo_0(8) = '0' and gpo_0(9) = '0') else serial2_CTS_3_3V1;
serial3_RX_combined <= serial3_RX_3_3V0 when (gpo_0(3) = '0' and gpo_0(10) = '0' and gpo_0(11) = '0') else serial3_RX_3_3V1;
serial3_CTS_combined <= serial3_CTS_3_3V0 when (gpo_0(3) = '0' and gpo_0(10) = '0' and gpo_0(11) = '0') else serial3_CTS_3_3V1;

```

RS232 voltage translation with a MAX232 IC UART transmission in Proteus and CCS

```

1  #include <main.h>
2
3  /* Main.c file for Project X
4   * Created: Tue May 6 2025
5   * Processor: PIC16F877A
6   * Compiler: CCS for PIC
7   */
8
9  #include <16F877A.h>
10 #use delay(clock=2000000) // 20 MHz crystal
11 #use rs232(baud=9600, xmit=PIN_C6, rcv=PIN_C7, ERRORS) // UART config
12
13 void main() {
14     char receivedData;
15
16     // Print initialization message
17     printf("UART Initialized\n\r");
18
19     while(TRUE) {
20         // Transmit data every 500ms
21         printf("Transmit Hello World\n\r");
22         delay_ms(500);
23
24         // Check for incoming data and echo it back
25         if(kbhit()) {
26             receivedData = getc(); // Read the incoming character
27             printf("Received data: [%c]\n\r", receivedData); // Print received data
28         }
29
30         delay_ms(500);
31     }
32 }
33

```

Use	Connections	Name	Description	Export	Clock	Base	End	IRQ
☒		clk_0 clk_in clk_in_reset clk clk_reset	Clock Source Clock Input Reset Input Clock Output Reset Output	clk reset <i>Double-click to</i> <i>Double-click to</i>	exported clk_0			
☒		hps_0 memory hps_io h2f_reset h2f_axi_clock h2f_axi_master f2h_axi_clock f2h_axi_slave h2f_lw_axi_clock h2f_lw_axi_master f2h_irq0 f2h_irq1	Arria V/Cyclone V Hard Proce... Conduit Conduit Reset Output Clock Input AXI Master Clock Input AXI Slave Clock Input AXI Master Interrupt Receiver Interrupt Receiver	memory hps_io hps_0.h2f_reset <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i>	 clk_0 [h2f_axi_... clk_0 [f2h_axi_... clk_0 [h2f_lw_a...			
☒		pio_0 clk reset s1 external_conne...	PIO (Parallel I/O) Intel FPGA IP Clock Input Reset Input Avalon Memory Mapped Slave Conduit	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> gpo_0	clk_0 [clk] [clk]	 0x0000_0080	0x0000_009f	IRQ 0 IRQ 0
☒		uart_0 clk reset s1 external_conne... irq	UART (RS-232 Serial Port) Int... Clock Input Reset Input Avalon Memory Mapped Slave Conduit Interrupt Sender	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> uart_0	clk_0 [clk] [clk] [clk]	 0x0000_0060	0x0000_007f	
☒		uart_1 clk reset s1 external_conne... irq	UART (RS-232 Serial Port) Int... Clock Input Reset Input Avalon Memory Mapped Slave Conduit Interrupt Sender	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> uart_1	clk_0 [clk] [clk] [clk]	 0x0000_0040	0x0000_005f	
☒		uart_2 clk reset s1 external_conne... irq	UART (RS-232 Serial Port) Int... Clock Input Reset Input Avalon Memory Mapped Slave Conduit Interrupt Sender	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> uart_2	clk_0 [clk] [clk] [clk]	 0x0000_0020	0x0000_003f	
☒		uart_3 clk reset s1 external_conne... irq	UART (RS-232 Serial Port) Int... Clock Input Reset Input Avalon Memory Mapped Slave Conduit Interrupt Sender	<i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> <i>Double-click to</i> uart_3	clk_0 [clk] [clk] [clk]	 0x0000_0000	0x0000_001f	