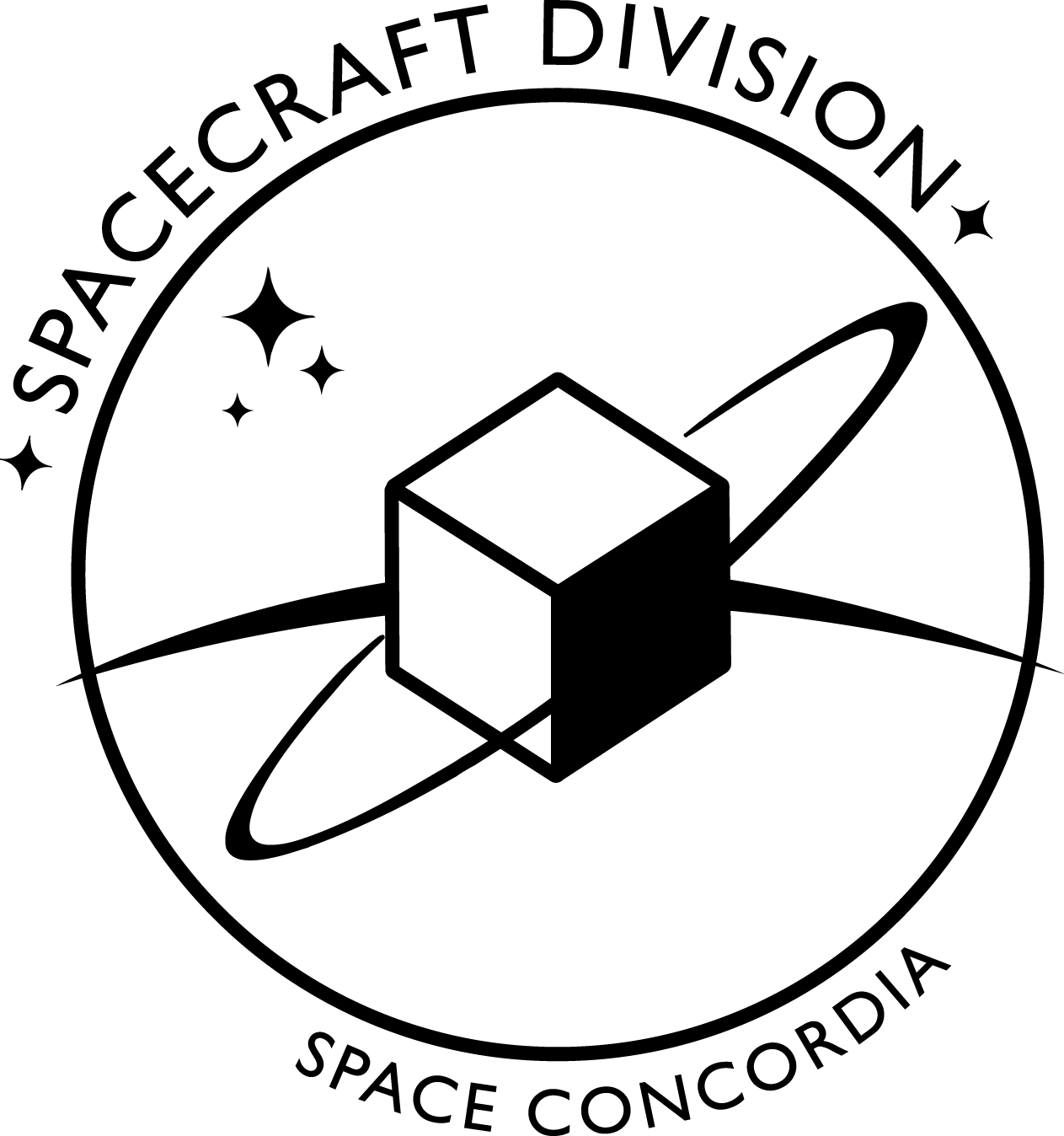
SCSD-LP-EPS-003-B

Serial Communication



Prepared by

[Gabriel Dubé](mailto:gabriel.dube@spaceconcordia.ca)

Electrical and Power System Lead

Jan 24, 2023

# **Revision History**

| **Date** | **Revision** | **Changes** |
| --- | --- | --- |
| 26-Jan-2023 | A | Initial Release |
| 16-Mar-2023 | B | Added RS-485 and some references |
|  | C | Added JTAG, SWD and ICSP debug protocols |

# **Table of Contents**

[**Revision History 2**](#_ad5thlvqzfhu)

[**Table of Contents 3**](#_z2p7d9s9al0w)

[**References 4**](#_ublz6vn9mh8u)

[**Abbreviations and Definitions 5**](#_z1bndbk4cgwt)

[**Introduction 7**](#_meu3f6nbbwk8)

[**Serial Communication 8**](#_3kwz5opnz7s6)

[UART 8](#_tc10rrylvlur)

[SPI 9](#_81nesm4e4gag)

[I2C 11](#_cl7acj7le)

[CAN 15](#_an8u48mb9mnx)

[USB 18](#_egachhupid44)

[LVDS 21](#_6njylk1x5zfk)

[RS-485 22](#_ysns6ocq8l0l)

[**Appendix A: Properties of Protocols 28**](#_1124of6nyci)

[**Appendix B: Differential Impedance 29**](#_3sdaaqlrkkl8)

# **References**

[1] CCP - CUBICS - ELEC - Design [internal document on Google Drive], “Breakout Boards”. Available from: <https://drive.google.com/drive/folders/1IolXB_eUfjbBWtSN64kAr6lkOlDXZvvk?usp=share_link>

[2] Electronoobs, [video] “PROTOCOLS: UART - I2C - SPI - Serial communications #001”. Youtube link: <https://www.youtube.com/watch?v=IyGwvGzrqp8&ab_channel=Electronoobs>

[3] BusProtection [research paper], “On-Orbit Implementation of Discrete Isolation Schemes for

Improved Reliability of Serial Communication Buses”. Available from: <https://drive.google.com/file/d/1PH2t4bVBQl9BAJ_0asnuh8AEZcqVkhuy/view?usp=share_link>

[4] Turnkey Electronic Engineering Solutions [software], “Saturn PCB Design Toolkit”. Free download: <https://saturnpcb.com/saturn-pcb-toolkit/>

[5] JLCPCB [website], “JLCPCB Manufacturing Capabilities”. Website link: <https://jlcpcb.com/capabilities/pcb-capabilities>

[6] Texas Instruments [website], “RS-485 basics: When termination is necessary, and how to do it properly”. Available from: <https://e2e.ti.com/blogs_/b/analogwire/posts/rs-485-basics-when-termination-is-necessary-and-how-to-do-it-properly>

[7] Altium Academy [video], “I2C and SPI on a PCB Explained!”. Available from: <https://youtu.be/eheh938ESU0>

# **Abbreviations and Definitions**

| **Terminology** | **Definition** |
| --- | --- |
| ADC | Analog to Digital Converter |
| ADCS | Attitude Determination and Control System |
| CAN | Controller Area Network |
| CDH | Command and Data Handling |
| CM | Common Mode |
| CSA | Canadian Space Agency |
| DM | Differential Mode |
| ESD | Electrostatic Discharge |
| GPIO | General Purpose Input/Output |
| LED | Light Emitting Diode |
| LVDS | Low-Voltage Differential Signaling |
| MCU | Micro Controller Unit |
| RTC | Real Time Clock |
| SC-FREYR | Fermentation R- Extraterrestrial Yeast R- |
| SPEAR-M7 | Flight computer of SC-ODIN |
| SPI | Serial Peripheral Interface |
| UART | Universal Asynchronous Receiver-Transmitter |
| USART | Universal Synchronous and Asynchronous Receiver-Transmitter |
| USB | Universal Serial Bus |

# **Introduction**

Serial communication refers to all the different communication protocols used in digital designs. These protocols all have different physical configurations, transmission speeds and communication ranges. They each require a specific number of connections and all differ in reliability factor. This document will mostly look at the physical layers of serial communication protocols, how to improve their reliability and layout considerations for each. This document will also look at some of the signal characteristics of each protocol, but will not go into detail. Only high-level functionality that is relevant to electrical. Multiple resources will be attached in the references section that deal with the software side of these protocols.

The main 6 protocols used for our Cubesat platform are UART, SPI, I2C, CAN, USB and LVDS. Some other protocols may be used for ground equipment, but these six protocols cover a good range of functionality that is required on a Cubesat, so the scope of this document will be limited to those. Note that LVDS was used in SC-ODIN only, and will most likely not be used on FREYR. With that being said, it is still a relevant protocol to learn about.

It is also important to note that a ground wire is ALWAYS required between all the devices connected by a communication protocol. It will be assumed every time the number of connections is mentioned.

# **Serial Communication**

## UART

UART is a device to device communication protocol, which means it only is able to handle communication between two devices. It is also a “Asynchronous” communication protocol, which means that there is no clock signal shared between both devices. The only two wires required by this protocol are a receiver and a transmitter wire (the ground connection is assumed). UART is one of the easiest protocols to implement on the electrical side. The few required connections and the usually low transmission speed make it not only easy to connect, but also generally reliable and noise resilient. Both the transmitter and receiver operate in common mode, so there is no complicated differential routing necessary.

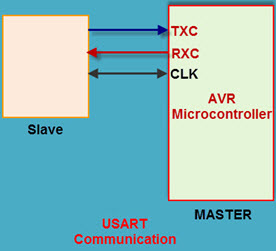
The physical connection of UART can be seen in Figure 1 below. RX is the transmitter signal and TX is the receiver.

## 

**Figure 1: UART Physical Layer**

As mentioned earlier, the advantages of this protocol are its simplicity, its reliability and the low amount of connection required. On the other hand, it is an overall slower protocol compared to others in this document and can only provide communication between two devices.

There also exists a variant of this protocol called USART. It works very similarly to the regular UART protocol, with the only difference being that it is synchronous as opposed to asynchronous. This means that it requires one extra connection: a clock signal. The RX and TX wires are also renamed to RXC and TXC to differentiate them from the regular UART signals. The physical layer of USART can be seen in Figure 2.



**Figure 2: Physical Layer of USART**

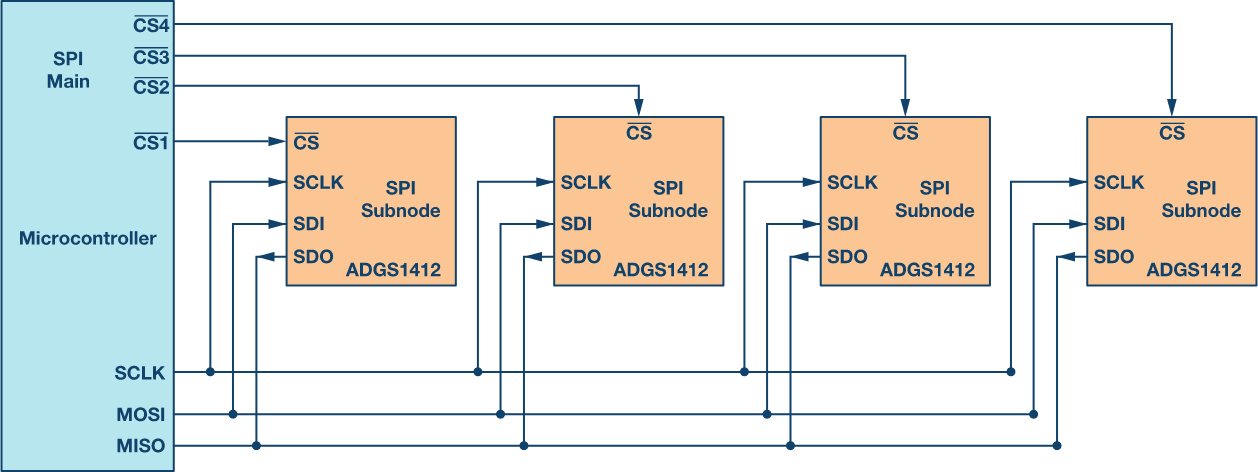
The main advantage of USART over UART is that by its synchronous nature, it allows for faster data rates. The only downside is that it requires an extra connection and some routing considerations given the higher speed.

## SPI

SPI is a hybrid between a device to device and bus type communication protocol. It retains the same principle as UART by having a transmitter and a receiver wire, but it also uses a clock signal and a number of chip select lines determined by how many devices are connected. The presence of a clock signal makes SPI a synchronous protocol, and it can run at a maximum of 12.5 MHz although it is usually used at much lower speeds, typically 1 MHz.

The physical layer of SPI requires the highest number of individual wires of any protocols in this document. These connections are:

* A transmitter, commonly called MOSI (master out slave in)
* A receiver, commonly called MISO (master in slave out)
* A clock, commonly called SCLK
* A chip select for every slave on the network

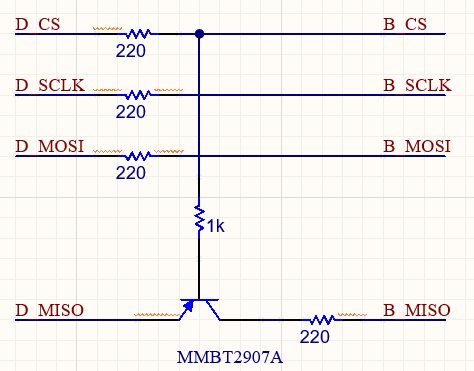


**Figure 3: Implementation of SPI with one microcontroller, the master,**

**and 4 SPI devices, the slaves.**

It is important to note here that the chip select lines are represented by and a number. The bar above, also called “bubble”, indicates that the pin is an active low. This means that to trigger the specific chip select of a slave, the master needs to pull the CS line to ground. Triggering this chip select will “wake up” the slave device and allow it to communicate on the MISO bus. This also means that when the master does not want to communicate with a specific slave, it needs to hold the CS line too high, essentially disabling the slave device. It is also important to note that it is not possible to communicate with more than one slave at a time, as they would interfere with each other on the MISO line.

A concern that arises with this network configuration is that in the case of a failure in one of the slave devices, the entire MISO line could be unusable. Depending on the type of failure experienced by the slave device, it is possible that it either completely grounds the MISO line, or even transmit random data, making the line virtually unusable. To prevent this from happening, a protection circuit must be implemented. Figure 4 shows this protection circuit.



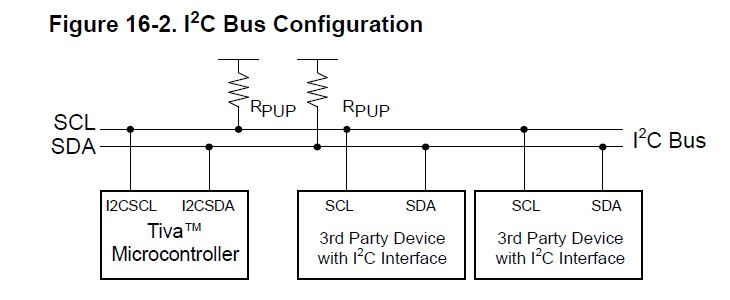
**Figure 4: SPI MISO bus protector**

The 220 ohm inline resistor will prevent too much current from passing between the devices on the data lines. The PNP transistor is configured in a way that follows the methodology of the chip select. When CS is high, the transistor is turned off, thus isolating the slave’s MISO line from the rest of the bus. When CS is pulled low, the transistor will go into active mode and will allow the slave to use the MISO line. This way, it is impossible for a slave to jam the MISO line as long as the master has access to the CS line.

## 

## I2C

I2C, commonly called “I squared C”, is a synchronous bus type communication protocol. I2C only requires two connections: SDA and SCL. SDA stands for Serial Data and SCL stands for Serial Clock, thus the synchronous feature of this protocol. Similar to UART, it is fairly easy to route since there are only two connections to make. The big difference comes from the fact that it is a bus type interface, which means that multiple devices can be connected to these same two lines, up to 128 devices. The other difference is that one of these wires is a clock signal, usually operating at 8 or 16 MHz, which will need some special considerations on the layout of its implementation.

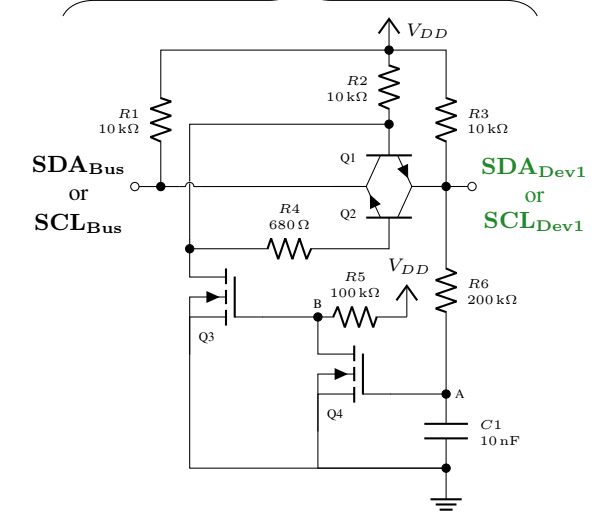


**Figure 5: Physical Layer of I2C**

There are two main things to note in Figure 5. The first one is that the Microcontroller is the master of the bus, and that the 3rd party devices are the slaves. The second thing to note is the presence of Rpup, which stands for pull-up resistor. Unlike most protocols in this document, the default level of the I2C lines is high, or VDD. This means that the master will send a 0 as a start bit, instead of the usual 1. The pull-up resistors ensure that the line remains at a high when no device is interacting with the bus. With that said, given the usual high resistance of these pullup resistors (>10K ohm), they will not interfere with the data sent on the SDA and SCL lines.

The question now becomes: how can the master communicate with only one slave if all slaves share the same data lines? To solve this issue, I2C uses “device addresses”. This means that the master stores a list of addresses, with one unique address assigned to each slave device. To begin communication, the master will send a start bit, essentially waking up all devices on the bus, and will then transmit an address string. If the address string received by the slave device matches its own address, it will then start receiving and sending the data on the SDA line. If a device receives an address that is not matching with its own, it will not interact with the bus, thus allowing the requested device to use the entire bus.

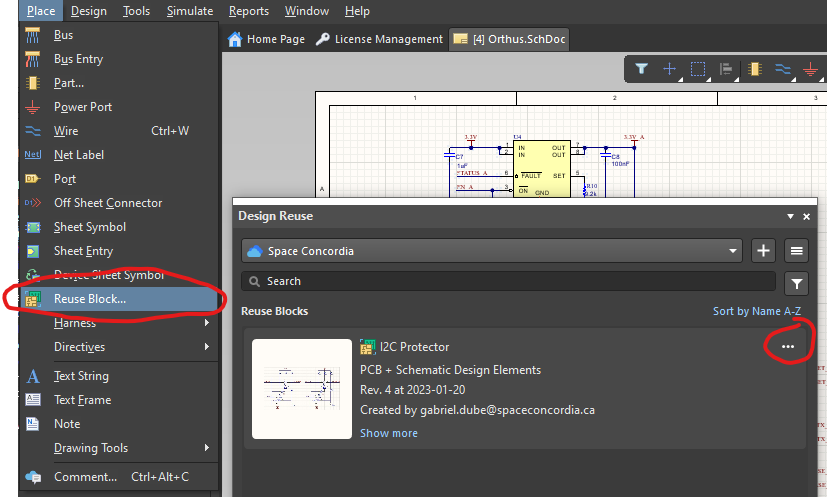
Just like SPI, there is a specific concern that arises in case of a slave device failure. In the case of I2C, the main concern is that if a device experiences a failure and holds either SDA or SCL to ground, the entire network becomes unusable. An easy solution could be to use a similar protection used for SPI, but this would mean adding an I/O line for every device, which defeats the entire purpose of a bus type protocol. Instead, an automatic isolation circuit is used. Reference [3] is a full research paper written on that particular circuit that goes in depth on its functionality, but this document will simply go over the main functionality. The protection or isolation circuit can be seen in Figure 6.



**Figure 6: I2C Bus Isolation/Protection Circuit**

During normal operation, the transistor pair Q1 and Q2 allow for bidirectional communication. R6 and C1 is basically an RC network that will follow the device voltage level. In the current configuration, it takes about 1.5 ms to charge or discharge the network. Considering the speed at which I2C operates, this will never turn on the protection during normal operation since the discharging is too slow. With that said, if a device were to hold its part of the bus to 0 for more than 1.5 ms, the capacitor will have time to discharge completely, which will trigger Q4, Q3 and thus, close Q2 and Q1. This essentially isolates the device from the network completely. It is fully automatic, and by its nature, it also resets automatically if the device is back to normal (a power cycle could get it back to normal).

There are also two big advantages when talking about implementation on an Altium schematic. The first one is that this protection circuit includes the necessary pull up resistors required by the I2C standard. This makes it practically impossible to forget them. The second advantage is that the Space Concordia library has this entire circuit, including the optimized PCB layout, in a “reuse block”. Figure 7 shows where to get access to the reuse block and how to place it. After placing the block to the schematic, importing changes to the PCB will import the already routed layout to your project, making it really easy to place and route.



**Figure 7: Reuse block in Altium**

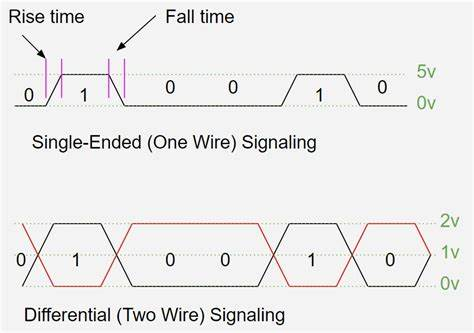
The first time you click on “Reuse Block”, nothing will be shown. It is important to click on the drop down menu and select “Space Concordia” instead of local. I2C Protector should appear in the list. All that is left to do is to click on the 3 dots and select “Place”.

For more information on how I2C and SPI should operate on a PCB, watch the video found in reference [7].

## 

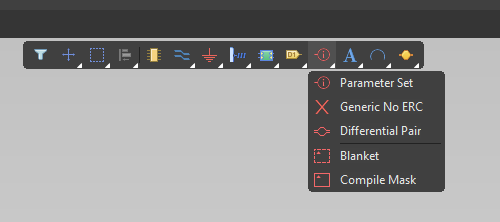
## CAN

Controller Area Network is a very robust and popular protocol, especially in the automotive industry. It is also a bus style protocol but unlike I2C, it is asynchronous, meaning there is no clock signal. It also differs from the last three protocols in the signal type: it is a differential signaling protocol. Differential signaling is the opposite of common mode signaling, since it does not use the ground as reference to express its voltage, but rather the difference between two wires. The two wires in question are the CAN H and CAN L wires, or CAN high and CAN low. This means that the device will see a voltage difference by comparing the H and L wire. In CM, it will see a voltage difference by comparing a wire to the ground reference. Figure 8 illustrates this difference.



**Figure 8: Common mode (single-ended) versus differential mode signaling.**

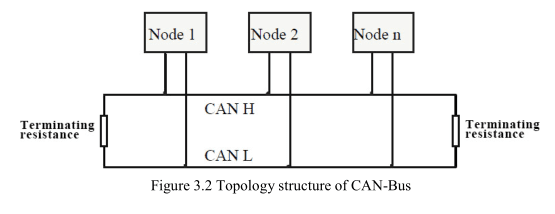
For PCB routing, this has a lot of considerations surrounding differential signaling. First, they need to be marked as “differential pairs” in the Altium schematics. Figure 9 shows this symbol on two CAN bus wires. It is important to note on that figure that instead of CAN H and CAN L, they are named CAN\_P and CAN\_N. This is because Altium will only recognize this specific syntax when naming differential pairs; one “\_P” and one “\_N”.



**Figure 9: Differential Pair Symbol in Altium**

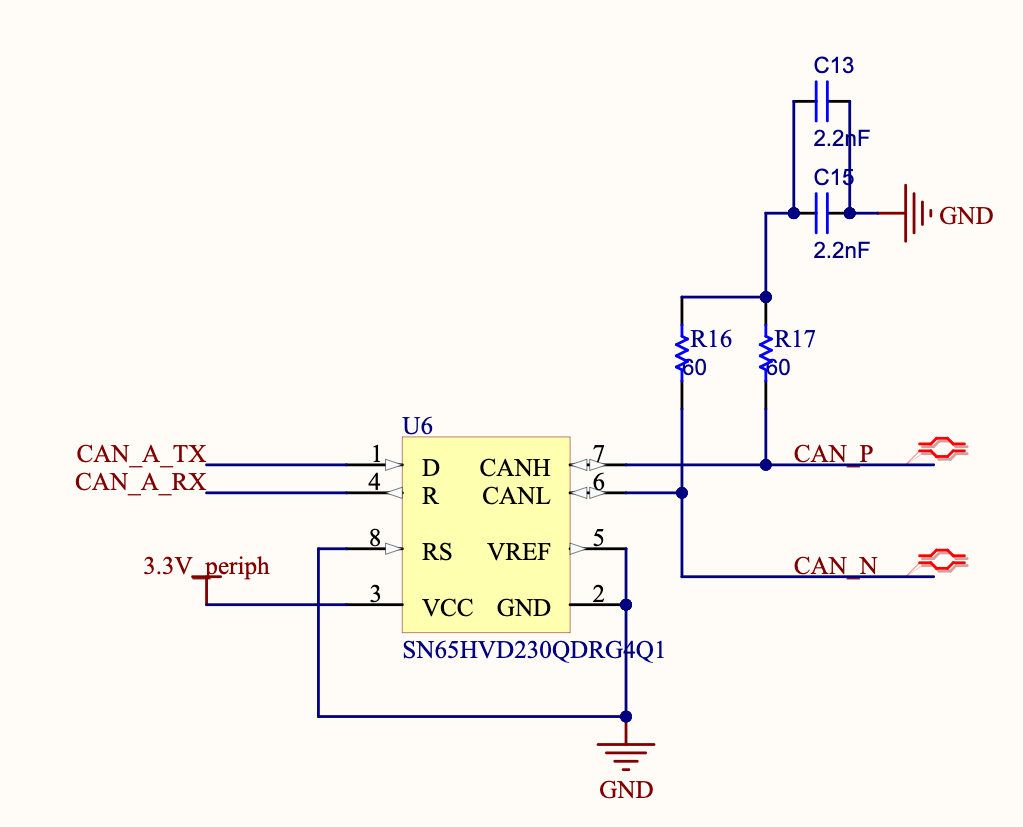
Once the schematic is set up properly, importing the changes to the PCB document will create the differential pair rule. This rule will need to be adjusted depending on the differential impedance of your protocol. For CAN bus, the target impedance is 120 ohms, with a tolerance of 10%. The referred impedance here is the impedance created by the wire’s self inductance and the capacitance between it and the ground plates. The real impedance, i.e the resistance, is negligible for copper wires and only the imaginary impedance is calculated. Obviously, this impedance is not to be calculated by hand every time given the complexity of the equations. Instead, there are many tools that can be used, notably the Saturn PCB tool (reference [4]). Using this tool, it is fairly easy to calculate the width, spacing and height of the differential pair traces and apply them as a rule. The appendix of this document contains a detailed guide on creating differential pair rules and calculating the values using the tool.

Another consideration that is unique to CAN bus is that not only do the traces need to have a target impedance of 120 ohm, but it also requires bus termination with the same target impedance. These terminations can be seen on Figure 10 and are typically a simple resistor with 120 ohm resistance. The physics behind bus terminations are complicated and cannot be explained in this document, but the main goal is to prevent signal reflections on either end of the loop. It would be suggested to look at reference [6].



**Figure 10: Typical CAN Bus Configuration**

The last consideration of CAN bus is that depending on the MCU used in a design, it might require an extra piece of hardware to properly implement it: a transceiver. The issue is that most MCU do not have CAN H and CAN L pins, but rather CAN RX and CAN TX. The RX/TX syntax can be recognized from the UART protocol, which is the exact opposite of CAN. This means that CAN TX and RX cannot simply be connected to CAN H and L, because one is CM and the other is DM. The way to fix this is by adding an external transceiver, which receives TX and RX as an input, and outputs H and L. Figure 11 shows the SN65HVD230QDRG4Q1, an automotive grade CAN transceiver, implemented on the CDH board of SC-ODIN. It is important to note 3 things on this figure: the conversion from CAN RX/TX to CAN P/N, the presence of the differential pair symbol and lastly, the bus termination. In this case, instead of a singular 120 ohm resistor, it was implemented using two 60 ohm resistors as well as two capacitors for noise reduction (they will act as a low pass filter to reduce the noise even more).



**Figure 11: CAN transceiver implementation in SC-ODIN**

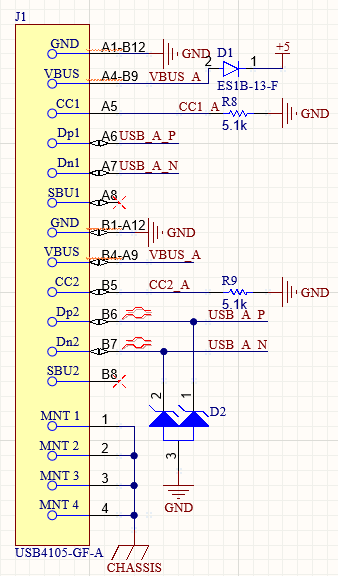
In conclusion, CAN bus definitely requires a lot of specific implementation and most of the time, extra hardware. There are also challenges on the signal side, since CAN bus does not have prevention against data collision, which can lead to delays in operation. With that said, it has a lot of advantages: high resilience to noise due to the DM signaling, up to a 100 devices on a single network and very long bus length depending on the data rate.

## 

## USB

USB is a widely known communication protocol due to its use in everyday electronics like computers and phones. It is a fairly simple, single ended and asynchronous communication protocol. Much like CAN bus, it only requires two wires: USB\_P and USB\_N. This also means that it is a differential signaling protocol and will require the exact same considerations that CAN bus does. The only difference is that the target impedance of USB, depending on the type, is around 90 ohms. As mentioned, USB has a lot of different types, mainly dictated by the speed it is implemented at. The most known are USB low-speed, full-speed, high-speed, 2.X and 3.X, with USB 4 being the fastest available USB on the market. For most Cubesat implementations, the preferred types are USB High-speed and USB 2.0 since slower speed generally equals an easier implementation and less concern for signal integrity.

Unlike CAN bus, USB is made to communicate between two devices only, meaning it is not a bus style protocol. It also does not require terminations, simply proper impedance matching on the routing. Another advantage of USB is that it not only can be used for data transfer, but also to supply power. The voltage level of the power supply can be adjusted, but it is generally a 5V power input, with the data lines alternating between 0 and 5V. Figure 12 shows a typical USB connector implementation. It also does not require an external transceiver, as most MCUs will have dedicated USB P and USB N pins. This can be seen on figure 13.



**Figure 12: USB connectors used on the CDH Breakout Board**

There are a few things to note about USB connector implementation. Firstly, the mentioned power input can be seen on the VBUS pin, and most connectors will have multiple VBUS pins to ensure current distribution across the different pins. The issue that arises with this is that there is a possibility that one VBUS back powers the second VBUS, potentially damaging the device connected to the port. Second, it can be seen that the Dp1/Dn1 and Dp2/Dn2 pins are all connected to the same data nets. This is specific to USB C given that it can be connected in two different orientations, and requires both data lines to be connected together to ensure it will work regardless of the orientation of the connector. It is also important to note that on both USB lines, there are suppressor diodes to protect against ESD, which is a major concern when dealing with connectors that will be constantly inserted and removed by users. Finally, the presence of resistors on the CC1 and CC2 pins is crucial when using the USB protocol since they dictate the receiver and transceiver relation between the two devices. If they are pulled high using a 5.1k resistor, it will be the transceiver, and if they are pulled down with the same value resistor, it will be the receiver. This also helps the devices understand the direction of current for power delivery.

## 

**Figure 13: USB pins on the SPEAR-M7**

One important thing to note on Figure 12 is the presence of zener diodes, also called suppressors (indicator D2). This is usually the only required protection feature for USB. The reason for these suppressors is for ESD protection (recommended to look it up). The ones selected here have a reverse breakdown voltage of 5.4V to ensure that any voltage spike above the usual operating voltage of 5V will be shunted to ground, protecting the device on the USB line. This is particularly relevant to USB because they usually involve a lot more user interaction, from plugging and unplugging the USB connector constantly. With the simple addition of the diodes, you can ensure that you will not “fry” the board during normal usage.

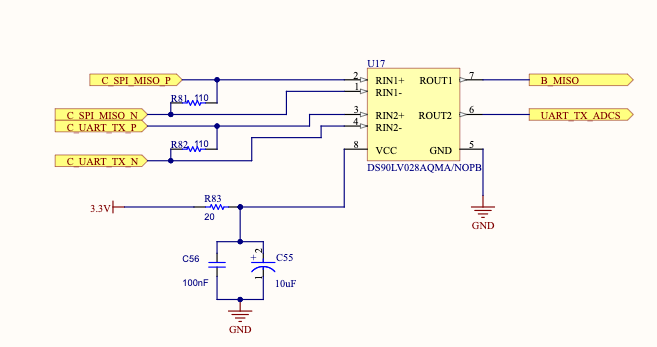
## LVDS

Last on the list is the LVDS protocol. As mentioned in the introduction, it is not a very commonly used protocol and is only really considered in special applications. During SC-ODIN, it was only used once for a specific purpose: transmitting data signals over a longer distance than usual. In Appendix B, there is a table that shows the maximum transmission length of some of the protocols mentioned in this document. It can be seen that some of them can only go as far as a few centimeters. Unfortunately, one of the PCBs in SC-ODIN exceeded that range from the main computer, making it unable to transmit certain data. To solve this problem, LVDS transceivers and receivers were used.

The basic principle behind LVDS is fairly simple: transform CM signals into DM, which increases their signal integrity and thus, can be transmitted to a further distance. Once it reaches its destination, all that is left to do is to convert the signal back into CM and send it to the targeted device. Figure 15 shows the implementation of an LVDS transceiver and Figure 14 shows a receiver.

## 

**Figure 14: LVDS transceiver on the CDH board**



**Figure 15: LVDS receiver on the CDH board**

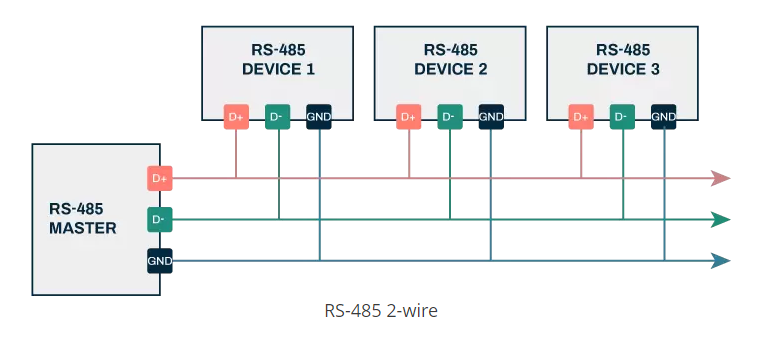
Few things to note on both figures: the transceiver takes the normal signals (on the right) and transforms them into their equivalent P and N signals. The 100 nF capacitors on the P and N lines serve as DC blocking capacitors, meaning the transceiver operates in AC transmission mode. On the receiver side, it is important to note the 110 ohm resistors in between the P and N lines to match the differential impedance of the transmission line (reference [6]).

LVDS definitely has its advantages for specific designs that require a range extension. With that said, it does come with a lot of challenges: all P and N pairs have to be routed differentially with 110 ohm target impedance, the implementation requires a lot of components and from experience, the transceivers and receivers tend to break during assembly, making it expensive and difficult to implement.

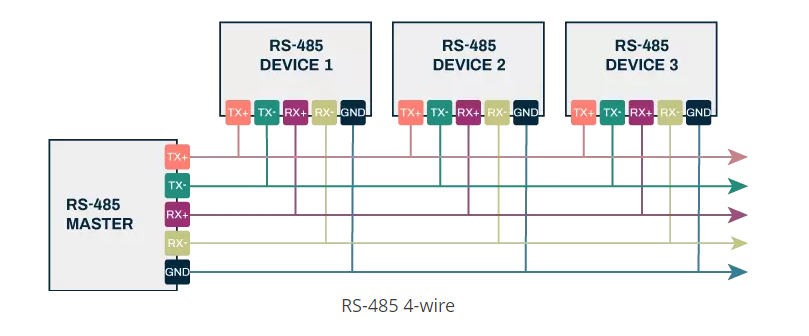
## RS-485

RS-485, unlike the previously mentioned protocols above, is not a protocol itself. It is rather a standard that defines the electrical characteristics of the drivers and receivers in a serial communication network. This explains the typical naming convention of these standards, RS, which stands for Recommended Standard. There exists a multitude of RS standards, all with different specifications and characteristics, but we will only look at RS-485 in this document. There are a few reasons for that: it is a widely used standard in the aviation and space industry, it offers very high data transfer rates for a short distance network and is fairly cheap and easy to implement. It excels particularly well in small local networks, which is great for small satellite applications. On top of all that, it is a differential signaling bus style standard, which makes it very resilient to noise and very reliable.

RS-485 offers two main network configurations: Half-Duplex and Full-Duplex. The half-duplex configuration offers the same speeds as full-duplex configurations, with the main difference being that full-duplex allows for simultaneous transmission and reception while half-duplex restricts to one at a time. Figure 16 and 17 show the two different network configurations.



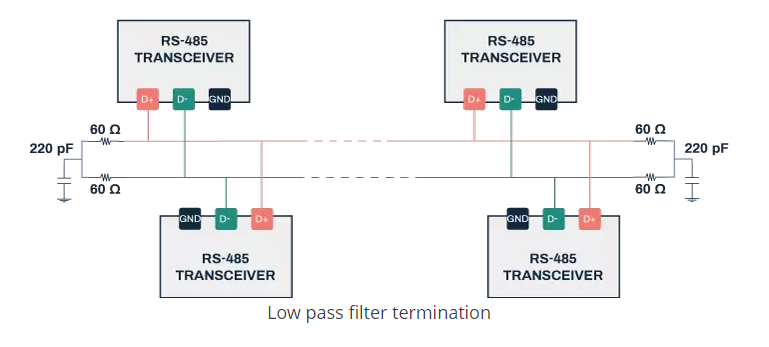
**Figure 16: RS-485 Half-Duplex Configuration**



**Figure 17: RS-485 Full-Duplex Configuration**

As can be seen on both figures, the full-duplex configuration requires more wires than the half-duplex one. It is also important to note that the signals are all differentially paired besides the ground signal.

Just like most differential signaling protocols, RS-485 will also require termination resistors on both ends of the bus. Figure 18 shows these termination resistors. It is important to note that these resistors are required to have a high level of precision (max 1%) given their criticality and that one resistor should be placed per differential pair when implementing a full-duplex network (reference [6]).



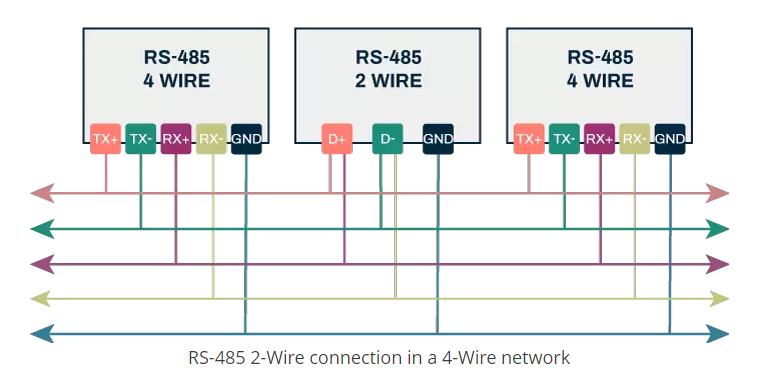
**Figure 18: Termination resistors for RS-485**

Much in the same way that the CAN Bus termination resistors were implemented (figure 11), the required resistance of the terminations is 120 Ω but was implemented as two 60 Ω resistors and a capacitor to act as a low pass filter.

A particularity of RS-485 is that when the bus is put into idle mode, i.e. when it is not being used for communication, all the differential nets are left floating since no device is driving the bus. This creates an unknown state on the data lines that could potentially be read as data by the devices. If this data triggers and starts bit, interrupt any other critical signal, it could result in issues and even a failure of a device. To prevent this, biasing circuitry should be added to force the lines into a specific state when left floating. This biasing circuit is simply a pullup resistor on all positive lines, and a pulldown resistor on all negative lines. The value of these resistors can be calculated using the equation below:

In this equation, R is the resistance value of both the pullup and pulldown resistor, Vcc is the node voltage and Rt is the termination resistance of the lines. With the right value, these resistors will basically reduce the “unknown” region of the data lines, reducing the chance of an unwanted event being triggered by the lines left floating.

Another great feature of RS-485 is its versatility. The standard can be implemented either as half-duplex, full-duplex, or both. Figure 19 shows both full and half duplex implemented on the same network.



**Figure 19: RS-485 implemented as both full and half duplex.**

The main reason why this is a great feature is because some devices might not be capable of supporting full-duplex implementation due to restricted pin availability or complexity constraints. If that is the case, the usual D+ can be connected to both RX+ and TX+ and D- can be connected to RX- and TX-. This will obviously remove the advantage of full-duplex for this particular device, but the rest of the network will still be able to use the full-duplex feature.

Now that the main configuration of the network is understood, it is time to look at the specific hardware implementation of it, more precisely the use of RS-485 transceivers. Most MCUs and devices will not have specific RS-485 drivers implemented, so the use of an external transceiver will be required. Fortunately, since RS-485 is a UART based protocol, any UART lines can be used to connect to the transceivers which will then generate the appropriate differential bus signals. A great example of a simple transceiver which will be used for SC-FREYR is the THVD1451DR ([datasheet](https://www.ti.com/lit/ds/symlink/thvd1410.pdf?HQS=dis-dk-null-digikeymode-dsf-pf-null-wwe&ts=1678924011099&ref_url=https%253A%252F%252Fwww.ti.com%252Fgeneral%252Fdocs%252Fsuppproductinfo.tsp%253FdistId%253D10%2526gotoUrl%253Dhttps%253A%252F%252Fwww.ti.com%252Flit%252Fgpn%252Fthvd1410)). Figure 20 shows the schematic diagram of this device.

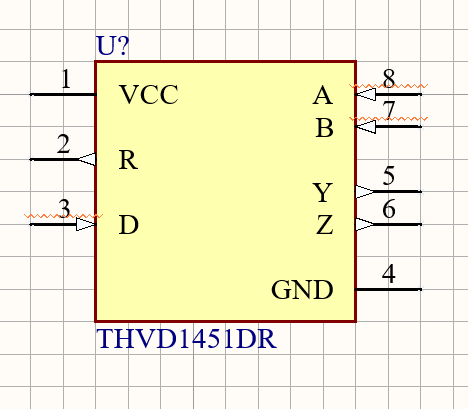
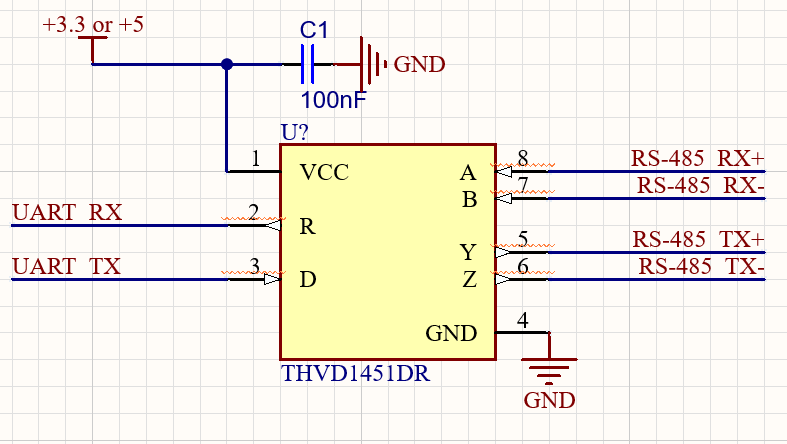


Figure 20: Schematic Diagram of the THVD1451DR

The device is fairly simple to implement as the schematic diagram suggests. VCC can simply be connected to a 3.3 or 5V power supply (with the usual 100nF decoupling capacitor, as usual) and GND to ground. D will need to be connected to the UART TX line and R will need to be connected to UART RX of the device being implemented on the network. A and B are the RX+ and RX- lines that can be seen in figure 17, while Y and Z are the TX+ and TX- lines. It is important to note that A will always be the positive signal, i.e. RX+, and Y will always be the positive transmission signal, i.e. TX+. Inverting these will create issues on the bus and make data unreadable. Figure 21 shows the full bus implementation using netnames to reflect the full-duplex implementation seen in figure 17.



**Figure 20: Implementation of the THVD1451DR**

The fact that most devices use UART or a form of UART makes this transceiver fairly versatile and easy to implement for most designs. Obviously, this involves putting one transceiver per device on the network (unless the specific device already has RS-485 integrated drivers), but given the size and ease of implementation of the device it should not be too difficult to add to most designs.

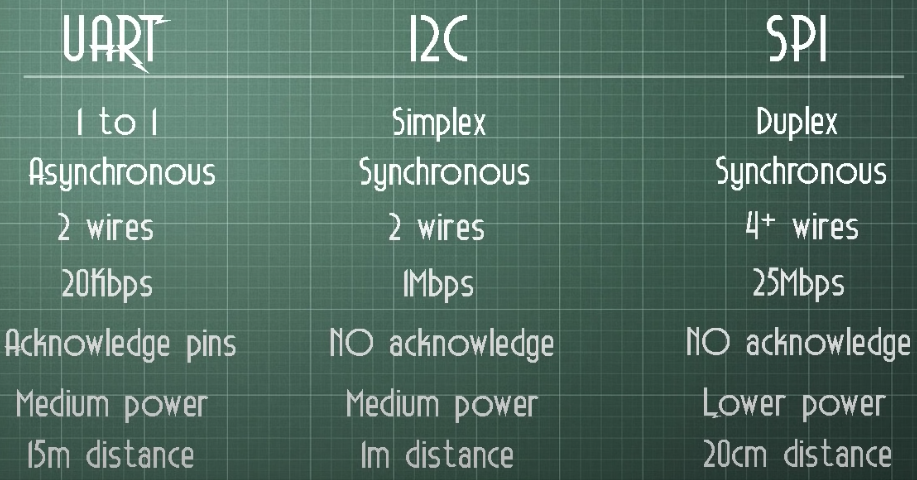
## JTAG

## SWD

## ICSP

# 

# **Appendix A: Properties of Protocols**



**Figure : Properties of UART, I2C and SPI protocols**

Reference [2]

# 

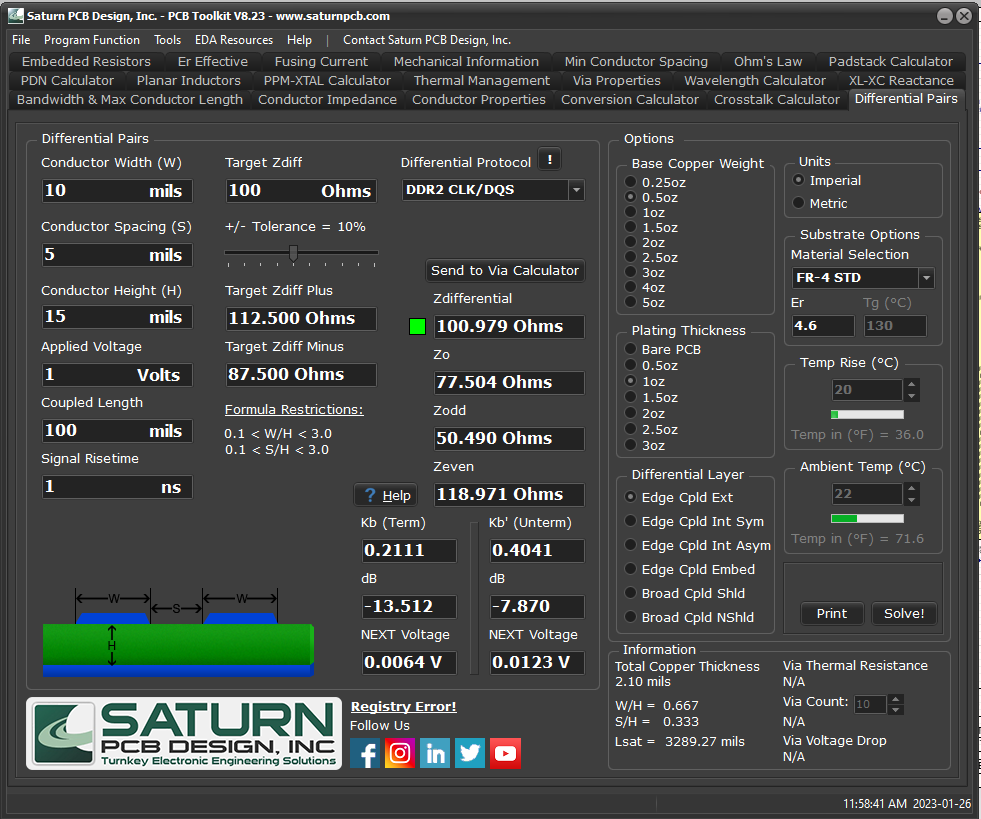
# **Appendix B: Differential Impedance**

This section will explain how to properly calculate target impedance, set the rule in Altium and adjust the ground polygons to respect the calculated values. Here are some rules to follow when routing differential pairs:

* Differential signals should always be routed first
* Differential traces always need to be on the top or bottom layer, never the inner layers.
* Ground layers are required as the first and last inner layers.
* No components should be placed too close to the differential pair (at least further than the calculated height in step 1).
* No components should ever be placed in between differential traces, with the exception of ESD protection diodes.

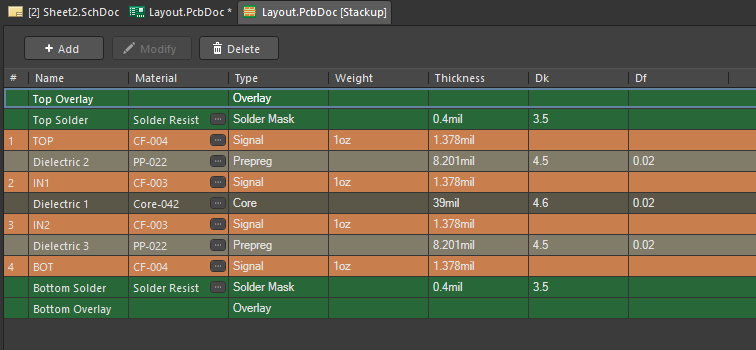
Step 1: Calculating target impedance values

The tool that will be used for this section is the Saturn PCB toolkit (reference [4]). When first opening the software, a bunch of options will be shown. The one of interest here is “Differential Pairs” and after selecting it, this page will appear.



This might look confusing at first, but there are only a few parameters that will be adjusted for our calculation. The first important parameter is the “Differential Protocol” drop down menu. Once clicked, a variety of protocols will appear, like USB, LVDS and many others. If the desired protocol appears, simply select it and it will automatically enter the “Target Zdiff” or target impedance for that particular protocol. If the desired protocol does not appear in the list, the Target Zdiff can always be set manually. The second parameter to verify is the Tolerance factor. In general, 10% is very fine but in some cases, a lower tolerance will be required (most of the time from the specific component’s datasheet). The third parameter to look out for is the “Zdifferential” parameter. This parameter will automatically adjust as you are changing the value of the traces. The last parameters that are important here and perhaps the most crucial are the Conductor Width, Spacing and Height. These values need to be adjusted manually by the user and the resulting impedance from that will be shown in the Zdifferential window. Trace width and spacing are pretty straightforward: the width of each individual differential trace and the spacing between the P and N wire. The height is the more complex one: it refers to both the height of the plane in reference to the ground plane underneath and to the spacing between the edge of the traces to the ground plane on the same layer as the traces. Whichever is closest is the value we will use for our calculations.

Let's start with the height of the plane. This value can be found on Altium by opening the PCB document and is located in Design > Layer Stack Manager. Once selected, this page will open.



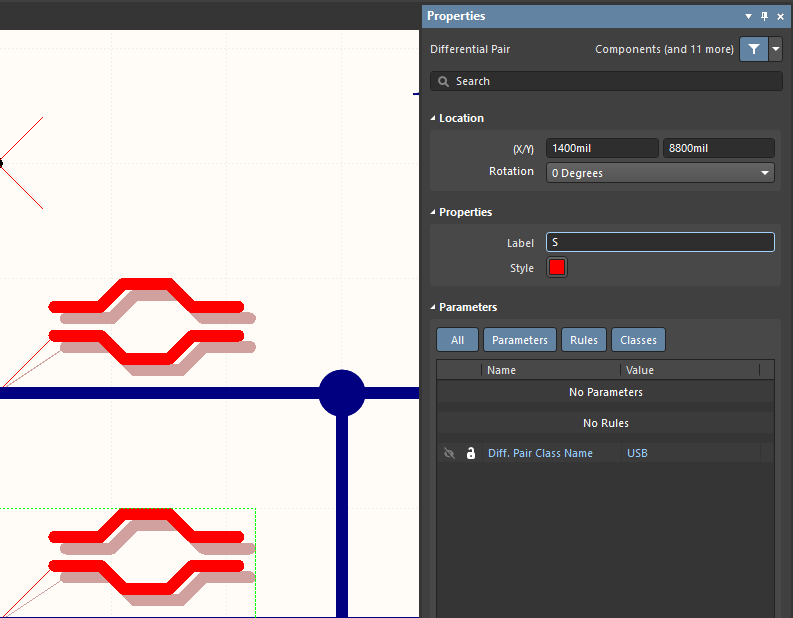
The values, number of layers and material will different depending on the layer stack chosen for the project, but the values required for impedance calculation will always be in the same location. Depending on if the trace is on the top or bottom layer, it will be required to look at the height of the dielectric between the top and first inner layer, or bottom and last inner layer. This will be the “Conductor Height (H)” parameter. It is important to ensure that the layer stack is final before performing this calculation. Once the value is found, simply enter it in the right window on the toolkit, press enter and observe the change in Zdifferential. If the differential impedance is not within the allowed range, or is too close to the max tolerance, adjust the W and S parameters until a reasonable impedance is achieved. As a general rule:

* Increasing spacing will “” impedance
* Increasing conductor width will “” impedance

Once the calculated impedance is satisfactory, save the W, S and H values somewhere easy to access and close the toolkit.

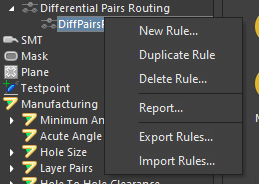
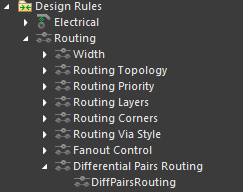
Step 2: Defining a differential pair rule in Altium

To properly set up a differential pair rule, it is first required to create a differential pair class. This will ensure that the rule will apply only to the specific protocols. Creating a class is fairly simple: simply go back to the schematics, double click on the differential pair symbol and locate the parameter section.

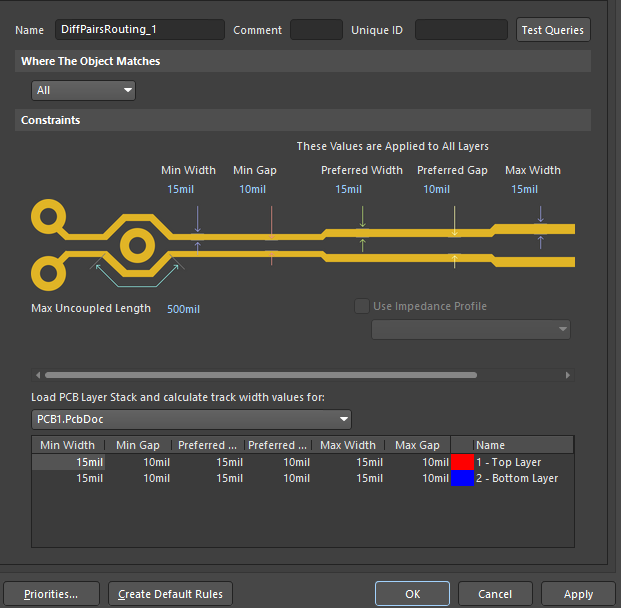


At the bottom of the section, click on “Add…” and select “Diff. Pair Net Class”. The parameter will be added to the list. Simply edit the “\*” and enter the name of the protocol (ex: USB in the image above).

The next step is to open the PCB document and import changes. This will import the differential pair class to the PCB, which will be important for applying our rule. To create the rule, go to Design > Rules… > Routing > Differential Pairs Routing and one default rule will already be there. Right click on “DiffPairsRouting” and select “New Rule…”.



A new rule will appear on the left menu called “DiffPairsRouting\_1”. Click on it and this menu will appear:

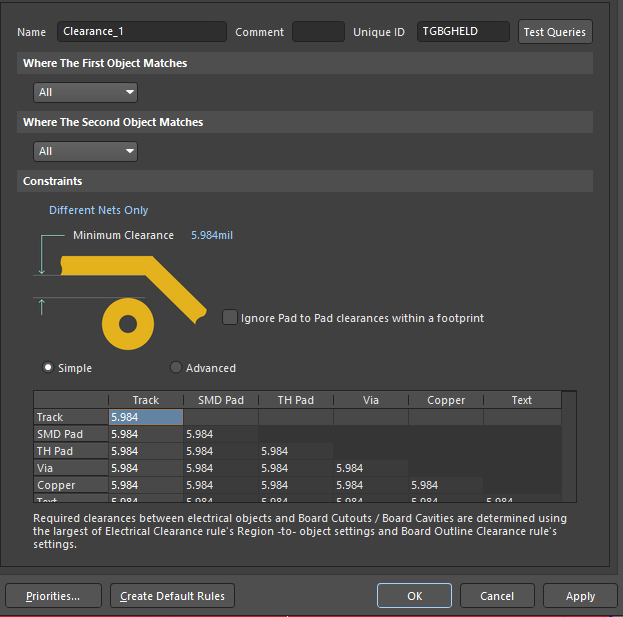


This is where the previously calculated values will need to be entered. First, rename the rule to “DiffPairsRouting\_[Name of protocol]”. Then, enter the calculated width in Min, Preferred and Max width slots. Then, enter the spacing in Min, Preferred and Max gap. The last step is to click on “Priorities…” and make sure the newly created rule is above the default one. Make sure to select “Apply” before closing the PCB Rules and Constraints Editor, and this step is done.

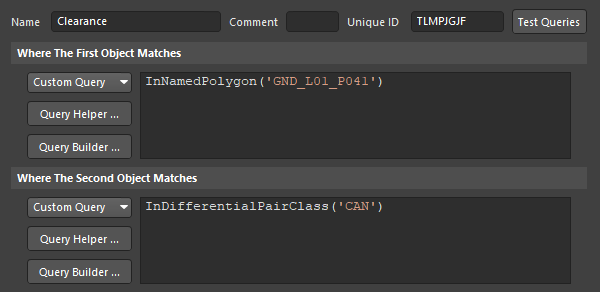
Step 3: Adjusting the ground polygon clearance on the same layer

This last step only applies if the default clearance rule of the PCB project is lower than the dielectric height of the layer stack. It should also be done at the very end of the design, when the layout is final and will not change aside from this adjustment. If that is the case, it will be important to define a new clearance rule to either match the height of the dielectric or simply use the top or bottom ground polygon as references for the height parameter. Before beginning this step, ensure that the traces in question are routed and that the top or bottom polygons are properly poured.

The first step will be to go back into the design rule editor and to go in the Electrical > Clearance section. There should only be one rule already present, and it is the universal clearance rule for every object on the PCB (check reference [5] for the value of this clearance). Once again, right click on the default rule and select “New Rule…”. This window will appear. Rename the rule to “DiffPairClearance\_[Name of protocol]”.

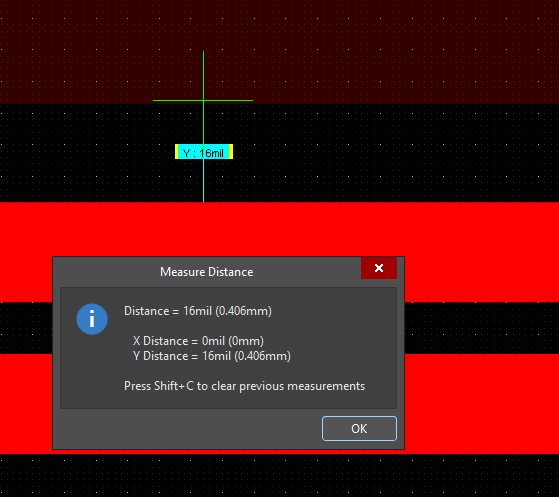


Then, click on the drop down menu under “Where The First Object Matches” with the option “All” by default and select “Custom Query”. Repeat the same for the second object. The following syntax should then be followed to write the rule:



The first object should always be the specific name of the ground polygon, in this case “GND\_L01\_P041”. Finding the name is very simple: simply go back to the PCB document and double click on the concerned polygon and copy the name. For the differential pair class, simply make sure it matches the name given in the first part of step 3. The only thing left to do is to edit the minimum clearance by entering the calculated height value. Make sure to press on “Apply” before closing the rule editor.

Finally, go back to the PCB document and go to Tools > Polygon Pours > Repour All. If everything was done correctly, the clearance around the mentioned differential pairs should be adjusted. To confirm that it was, it is possible to measure manually the distance between the edge of the traces and the top polygon. To do so, go to the PCB document, zoom in on the trace in question and press CTRL M. Click on the two points you want to measure and it will appear on the screen. To remove the marking, simply press CTRL C.



# 