



Norwegian University of  
Science and Technology

## **Project Thesis TFE4580**

Automotive embedded system  
implementation using System-on-Module

**Trym Sneltvedt**

December 2019

Supervised by Bjørn B. Larsen



# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Background . . . . .	4
1.1.1	Revolve NTNU . . . . .	4
1.1.2	The vehicle control unit . . . . .	4
1.1.3	Analysis of 2019 system . . . . .	4
1.2	Scope . . . . .	7
<b>2</b>	<b>Theory</b>	<b>7</b>
2.1	Ball Grid Array PCB layout . . . . .	7
2.2	CAN-FD . . . . .	8
2.2.1	CAN-FD bus utilization . . . . .	8
2.2.2	Maximum utilization . . . . .	9
2.2.3	CAN-FD utilization . . . . .	9
2.3	Differential signalling . . . . .	10
<b>3</b>	<b>Method</b>	<b>11</b>
3.1	System-on-Module . . . . .	11
3.2	Partitioning interfaces . . . . .	12
3.2.1	CAN-FD buses . . . . .	13
3.2.2	Ethernet for telemetry . . . . .	14
<b>4</b>	<b>Results</b>	<b>16</b>
4.1	Hardware design . . . . .	16
4.2	CAN-FD buses . . . . .	16
4.3	Ethernet . . . . .	17
4.4	Further work . . . . .	17
<b>5</b>	<b>Discussion</b>	<b>17</b>

5.1	PCB design . . . . .	17
5.2	PCB production . . . . .	18
5.3	PCB assembly . . . . .	18
5.4	PCB testing . . . . .	18
<b>6</b>	<b>Conclusion</b>	<b>19</b>
	<b>References</b>	<b>19</b>

# 1 Introduction

## 1.1 Background

### 1.1.1 Revolve NTNU

Revolve NTNU is a student organization dedicated to constructing electric formula cars for the international Formula Student (FS) competitions. In the course of one year, an Electric Vehicle (EV) is designed, constructed and tested before entering multiple competitions, facing off against comparable vehicles and teams from other universities from around the world.

The combination of short development time and a team consisting of mostly new, inexperienced members makes for several interesting problems. This report will focus on a specific embedded system, the Vehicle Control Unit (VCU), and the issues encountered when developing it. As the EV is almost exclusively made from custom parts, the different systems on the car depends heavily on each other. This means that changes done to the VCU will invariably affect the rest of the vehicle and vice-versa.

### 1.1.2 The vehicle control unit

The vehicle control unit is the central control system of the EV. It is responsible for handling sensory data from the vehicle and the driver, and then output fitting data to the motor controllers (*inverters*) for each electric motor. The VCU is only allowed to send set points to the inverters when the EV is in *Drive-Enable mode*. Transitioning into this mode requires action from the driver.

The team member responsible for the VCU is not responsible for the implementation of the control algorithms that takes vehicle and driver input and outputs motor set points. The control algorithms performs Torque Vectoring (TV), a technique for intelligently varying the torque on each wheel. This increases the maneuverability of the EV, especially in corners. The VCU is responsible for providing an environment for the control algorithms to run.

It is necessary that the VCU conforms the to FS competition rules. Formula Student Germany (FSG) is the largest competition, and their rules are considered standard [8].

### 1.1.3 Analysis of 2019 system

The EV made by Revolve NTNU during the 2019 season is called Nova. A simple overview of the systems and their interactions can be seen in figure 1.

In the 2019 season, Revolve NTNU made their newest EV, Nova. (see figure ??).

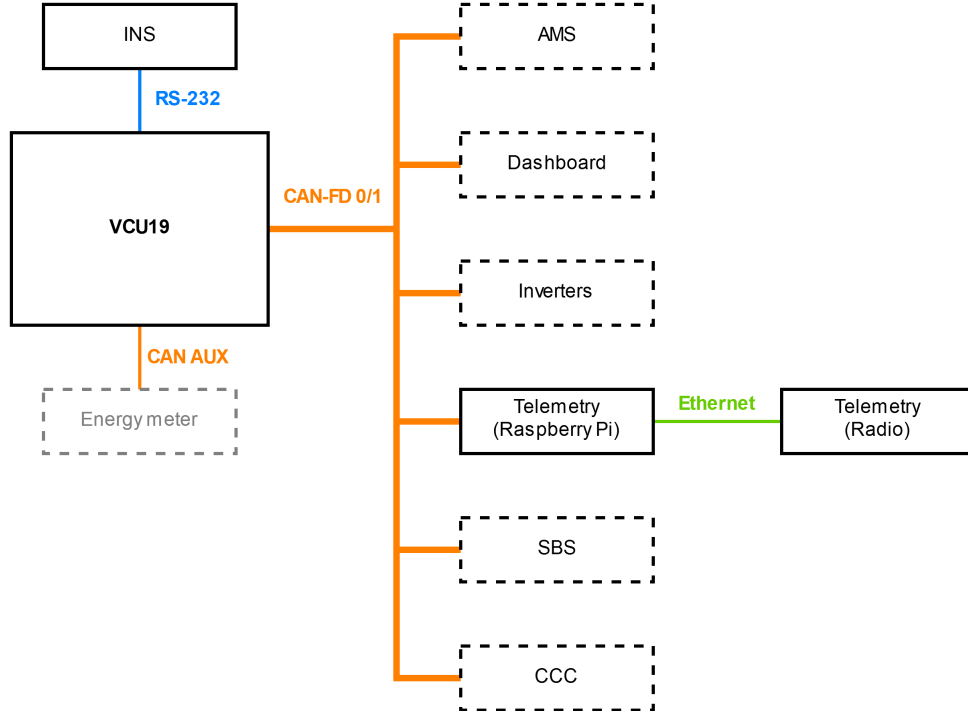


Figure 1: Author’s rendition of embedded system structure in Nova. Systems developed by other team members have dotted borders.

To fulfill the requirements of the TV control algorithm, Vehicle Control Unit 2019 (VCU19) was designed around a new processing platform, the Xilinx Zynq-7000 [12]. It is a System-on-Chip (SoC) with a dual core ARM Cortex A9 processor and embedded Field Programmable Gate Array (FPGA).

Advanced processing units like the Zynq-7000 are typically only available in Ball Grid Array (BGA) packages, meaning packages with all pins placed in a grid pattern on the bottom. This makes assembly and debugging harder, a serious concern considering the rapid development methodologies necessary in Revolve NTNU. A 3D render of VCU19 can be seen in figure 2.

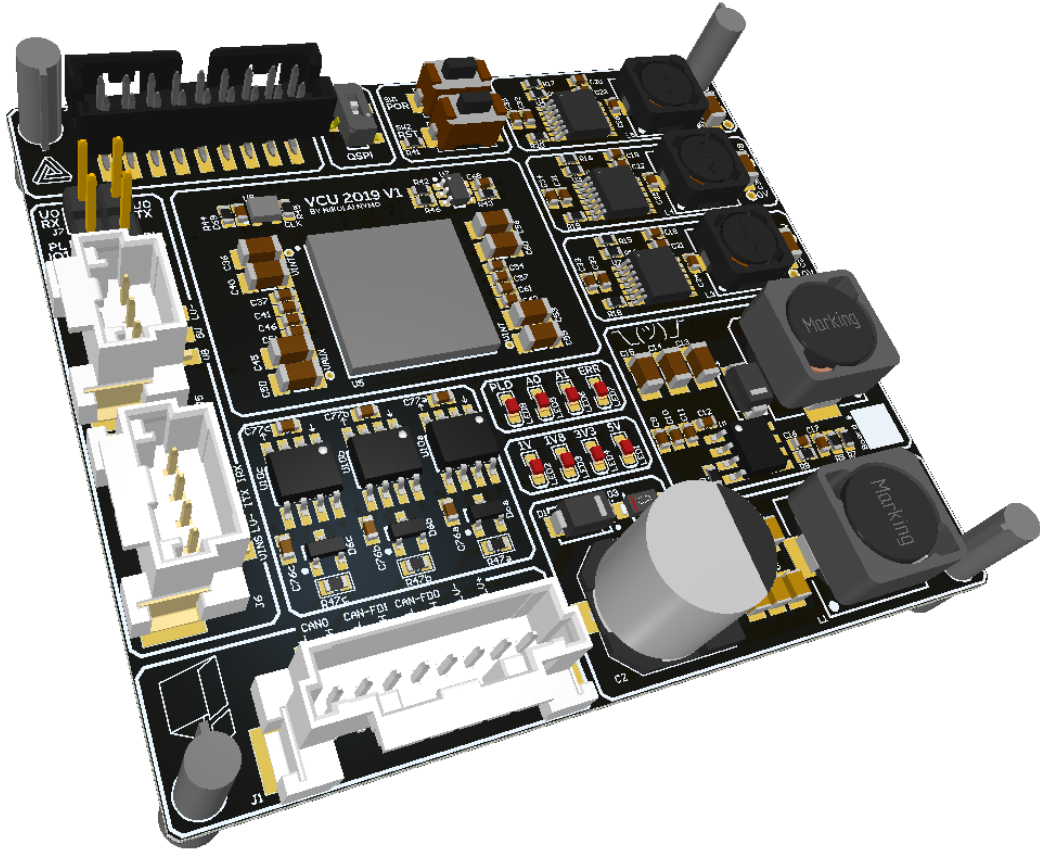


Figure 2: VCU from 2019 season. It was designed by team member Nikolai Nymo, VCU responsible in 2019.

Another issue encountered was the high load on the Controller Area Network Flexible Data-rate (CAN-FD) buses. Much of the work in Revolve NTNU is directed at analyzing data from the EV. The data is transmitted from the vehicle to a base station nearby using a proprietary wireless communication system from Radionor AS [3]. This is called the *telemetry* system.

The radio must be used through an Ethernet interface and there are no other embedded systems on the EV equipped with Ethernet. This was solved by connecting a Raspberry Pi [7] to the two CAN-FD buses of the car using two PCAN-FD USB dongles [9]. The Raspberry Pi is equipped with an Ethernet jack, allowing it to gather data from the CAN-FD buses and transmit it to the radio.

This solution introduces two issues. Firstly, this means that all the data that are to be sent over telemetry has to be present on the buses. As a result, the buses operate at nearly maximum capacity, introducing the risk of bus congestion (i.e. loss of messages).

The other issue is that the telemetry translation adds weight to the EV. The accumulated weight of the Raspberry Pi and the two dongles amounts to almost 200g. Revolve NTNU works hard to reduce the weight of the vehicle as it increases the possible acceleration and allows for shorter lap times.

## 1.2 Scope

This report covers the design, assembly and testing of Vehicle Control Unit 2020 (VCU20). The new design reduces the hardware complexity of the system (while keeping the performant Zynq-7000 SoC platform) and reduces both total vehicle weight and CAN-FD bus load.

## 2 Theory

### 2.1 Ball Grid Array PCB layout

BGA refers to a packaging type used for surface mount ICs. BGA packages feature all interconnects on the bottom of the package and allows for a higher number of connected pins than with pins on the edges of the package, see figure 3.

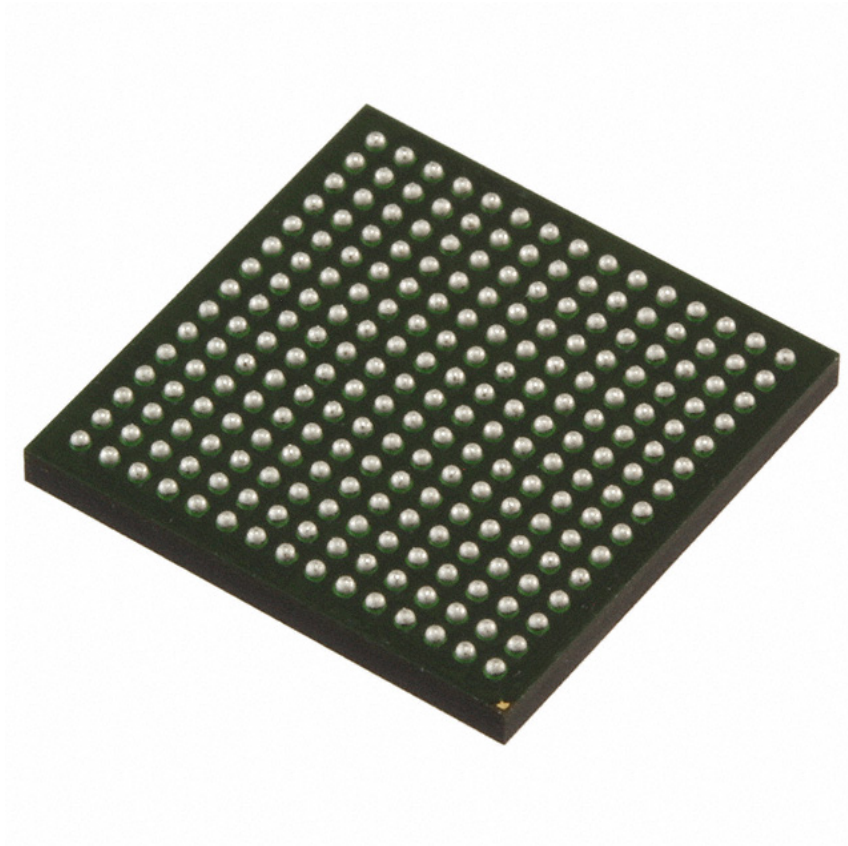


Figure 3: CLG255, a 15-by-15 pin BGA package used by some SoCs in the Xilinx Zynq-7000 series. Illustration image taken from the XC7Z010-2CLG225I product page on [www.digikey.com](http://www.digikey.com) [16].

As the pins are hidden when the package is mounted on a PCB, BGA packages must be soldered by other methods than classical hand-soldering. Reflow soldering is a popular method for soldering BGA's where the entire package and adhering PCB are heated to



the point where all the solder melts. This is typically performed in a dedicated reflow soldering oven, but can also be performed using a hot-air gun or similar.

## 2.2 CAN-FD

The different embedded systems on the vehicle communicate with each other using CAN-FD. Controller Area Network (CAN) is a network bus developed by Bosch. It has been the de-facto standard communication platform for automotive applications since the mid-1990's. An improved version, Controller Area Network Flexible Data-rate (CAN-FD) was released by Bosch in 2012, it differs from CAN in that the body of each message (called *frames* for CAN) are transmitted at a higher frequency than the header and the allowed frame size increases from 8 to 64 bytes. This is the communication bus currently used at Revolve NTNU.

CAN and CAN-FD is a one-to-many bus system where each frame is given an identifier which determines the priority of the frame. When the bus is free, all connected devices can begin transmitting their frame, but the frame with the highest priority will take precedence on the bus. When a frame is transmitted on the bus, any connected device are able to read it. When the frame has successfully been transmitted the bus is again free and a new transmission can begin.

An overview of the CAN-FD base frame format is seen in table 1.

Table 1: CAN-FD frame format.

Field	Name	Size (bits)
<i>SOF</i>	Start-of-frame	1
<i>ID</i>	Frame identifier	11
<i>RTR</i>	Remote transmission request	1
<i>IDE</i>	Identifier extension bit	1
<i>r0</i>	Reserved bit 0	1
<i>DLC</i>	Data length code	4
<i>Df</i>	Data field	0-512
<i>CRC</i>	Cyclic redundancy check	15
<i>CRCdel</i>	CRC delimiter	1
<i>ACK</i>	Acknowledge	1
<i>ACKdel</i>	ACK delimiter	1
<i>EOF</i>	End-of-frame	7

### 2.2.1 CAN-FD bus utilization

Almost all messages sent over the CAN-FD bus are sent at a regular interval. The size of the different messages are also known, this means that we know how long each message will occupy the bus for. These facts means that the frames on the bus can be modelled in the same way as tasks in a Rate-Monotonic Scheduling (RMS) system, known from real-time theory. This means the standard formula for utilization can be used when analyzing CAN and CAN-FD buses. The equation for utilization  $U$  on a bus is given in equation 1.

$$U = \sum_{i=1}^N \frac{C_i}{T_i} \quad (1)$$

$C_i$  is how long task  $i$  takes to execute (execution time) and  $T_i$  is how often the task is set to execute (execution period). In our case this is translated to the time frame  $i$  is active on the bus and the period at which it should be transmitted.

### 2.2.2 Maximum utilization

We can use the *Utilization-based schedulability test* [1] to determine whether all deadlines are met for a set of tasks. The criteria can be seen in equation 2. It is important to note that this test is sufficient but not necessary. This means that a passing test guarantees all deadlines are met but a failing test does not guarantee missed deadlines.

$$U = \sum_{i=1}^N \frac{C_i}{T_i} \leq N \left( 2^{\frac{1}{N}} - 1 \right) \quad (2)$$

$N$  denotes the number of different tasks being scheduled. Since the number of different messages on the CAN-FD buses is quite high, we might as well look at what happens to the criteria when  $N$  goes towards infinity.

$$\lim_{N \rightarrow \infty} N \left( 2^{\frac{1}{N}} - 1 \right) = \ln 2 \approx 0.693 \quad (3)$$

From equation 3 we can deem that as long as the bus load on each of the CAN-FD buses are below 69.3%, no messages are lost. This does of course not mean that the vehicle becomes unusable, figure 6 shows that the load was above the the limit for both buses, and the vehicle worked fine. The embedded systems are naturally made to be resilient to frame loss.

However, the limit serves as a good target for max bus load, and we will work towards lowering bus load to below this limit.

### 2.2.3 CAN-FD utilization

From the RMS utilization equation we know that we need the execution time  $C$  and period  $T$  for each task. While the period for each CAN-FD message is easy to determine, the execution time takes slightly more effort. From now on we will call the execution time for a message  $m$  the *transmission time* and we will denote it  $C_m$ . For a standard CAN message it would be sufficient to divide the number of bits in the message by the operating frequency of the bus. This is made slightly more difficult as CAN-FD operates on two different frequencies, one for arbitration (header) transmission and one for data transmission. We can split look at the total transmission time as a sum of the transmission time for each of the frequencies, see equation 4.

$$C_m = C_s + C_f \quad (4)$$

$C_s$  is the transmission time for the message header. It is independent of frame size and the calculation can be seen in equation 5.

$$C_s = \frac{(SOF + ID + r1 + IDE + EDL + r0 + \frac{BRS}{2} + \frac{CRCdel}{2}) \cdot 1.2}{t_x} + \frac{ACK + DEL + EOF + IFS}{t_x} \quad (5)$$

$C_f$  is the transmission time for the frame data, sent at a higher speed than the header. It is calculated as in equation 6 where  $D_f$  is the data size in bits.

$$C_f = \frac{(D_f + \frac{BRS}{2} + ESI + DLC + \frac{CRCdel}{2}) \cdot 1.2 + CRC + BS}{t_y} \quad (6)$$

$SOF$ ,  $ID$ ,  $IDE$ ,  $EDL$ ,  $r0$ ,  $r1$ ,  $BRS$ ,  $CRCdel$ ,  $ACK$ ,  $DEL$ ,  $EOF$ ,  $IFS$ ,  $ESI$  and  $DLC$  denotes bitfields of fixed size in the CAN messages, i.e. they are constant.  $t_x$  denotes the period of the arbitration frequency and  $t_y$  is the period for the data frequency. 1.2 is a factor for ensuring worst case behaviour.  $D_f$  is the actual data that is transmitted in the frame and is almost the only size that can change in equations (5) and (6).

Because of error detection mechanisms defined in the CAN-FD protocol, the transmission time for payloads larger than 16 bytes is different from payloads smaller than 16 bytes. For payloads smaller than 16 bytes the Cycling Redundancy Check (CRC) is 17 bits and Bit Stuffing (BS) is 5 bits. For payloads larger than 16 bits,  $CRC$  rises to 21 bits and  $BS$  to 6 bits.

This is all that is needed to calculate the worst case load of a CAN-FD frame [2].

## 2.3 Differential signalling

When dealing with high-speed, low-power signals in electronic systems, minimizing noise is of the utmost importance. One technique for dealing with this is *differential signalling*. Simply put, it means driving two conductors instead of just one. Single-ended signalling, the conventional method, means that signalling lines are referenced to a common ground. In differential signalling, two lines are used for each signal line. These lines are referenced to each other instead of to a ground shared by all signals. This serves to remove noise that might be present on the ground plane and, when the two conductors are placed close to each other, excellent protection against external interference. When the lines are close to each other, electromagnetic fields that normally would introduce noise to the signal will optimally affect both conductors by the same amount, and since we only look at the difference between the two lines, the noise is elegantly subtracted away.

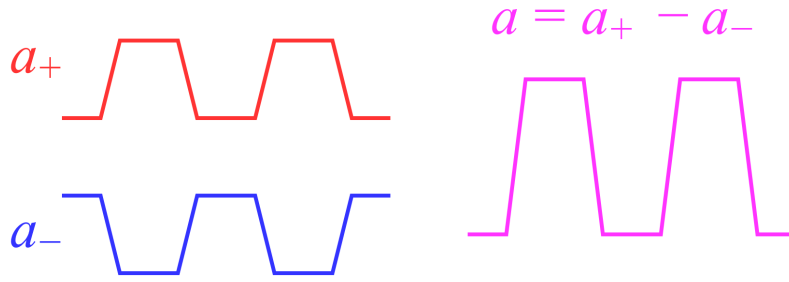


Figure 4: Differential signalling illustrated. The signal is the difference between the signals on line  $a$  and  $b$ .

### 3 Method

This section covers the choices made during the design of VCU20. Each subsection takes on a different challenge encountered in the 2019 season, see section 1.1.3.

#### 3.1 System-on-Module

The Zynq-7000 platform provided a sufficient platform for running the TV algorithm and control loop. However, the Zynq-7000 SoC product line are only available in BGA packages. This introduced unwanted complexity to the design, both in term of requirements to the PCB manufacturing, and to the assembly of the circuit boards which is a process that normally is performed by team members at the electronic workshop at Revolve NTNU's headquarters.

In regards to the PCB production process, the BGA package of the SoC adds two kinds of complexity to the design. The first is the need for very tight tolerances. To be able to properly extract all necessary signals from the SoC, a trace width of 0.1mm is needed. Secondly, the circuit board requires a higher number of layers. This is mainly because the SoC requires multiple different supply voltages.

Both these requirements increase the price of PCB production drastically. Although PCB production houses have increased in number over the last few years, thereby driving the manufacturing cost down, special production techniques like more than 4 layer PCBs and 0.1mm trace widths still introduces high costs to production runs. Table 2 shows a cost comparison of the different tolerances and layer numbers.

Table 2: Prototype PCB production cost depending on tolerance and number of layers. Data gathered from PCBWay's online instant quotation tool[14].

Layers	Tolerance	Quantity	Cost
4	0.20mm	5	49 USD
8	0.10mm	5	388 USD

The solution to this issue is to employ as System-on-Module. Instead of mounting the SoC directly on the VCU20 PCB, we can purchase off-the-shelf SoC modules. These modules consists of a small PCB featuring a SoC and some peripherals like external DDR RAM and non-volatile flash memory. The modules interconnects with the VCU PCB by one or more PCB connectors which are easier to route. The downsides to using these modules is mainly the cost. The module utilized for VCU20, Enclustra Mercury ZX5 with Zynq-7015 (figure 5), has a unit price of 318USD. However, as they are modular, they can easily be reused between seasons. Reusing soldered ICs is not common practice at Revolve NTNU, so this should mitigate the high entry cost at least to some extent.

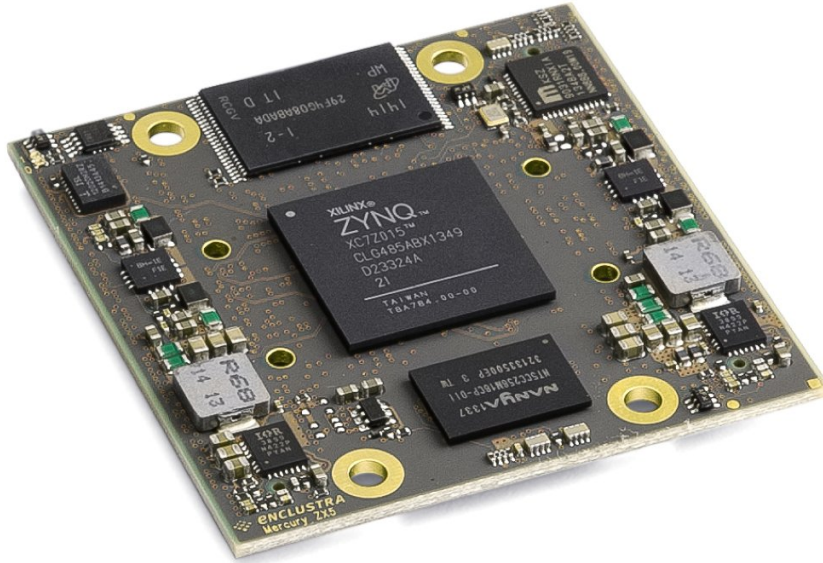


Figure 5: Enclustra Mercury ZX5 SoM. Product image retrieved from Enclustra website [4].

### 3.2 Partitioning interfaces

As the VCU is an integral part of the EV, issues regarding the vehicle are also issues for the VCU. This section is dedicated to some of the issues encountered during last season, which hopefully can be solved by improving the design of the VCU.

In Nova, the main communication channel for the embedded systems were two CAN-FD buses running at a base frequency of 1MHz and an data transfer frequency of 4MHz. A system overview is displayed in figure 1.

There are several points where this system can be improved. They will be discussed in detail in the following subsections.

### 3.2.1 CAN-FD buses

A trace of the load on both CAN-FD buses during normal operation can be seen in figure 6. Note the increase in load on bus 1 halfway into the trace. This is caused by the car entering Drive-Enable (DE) mode where the VCU starts transmitting set points to the inverters.

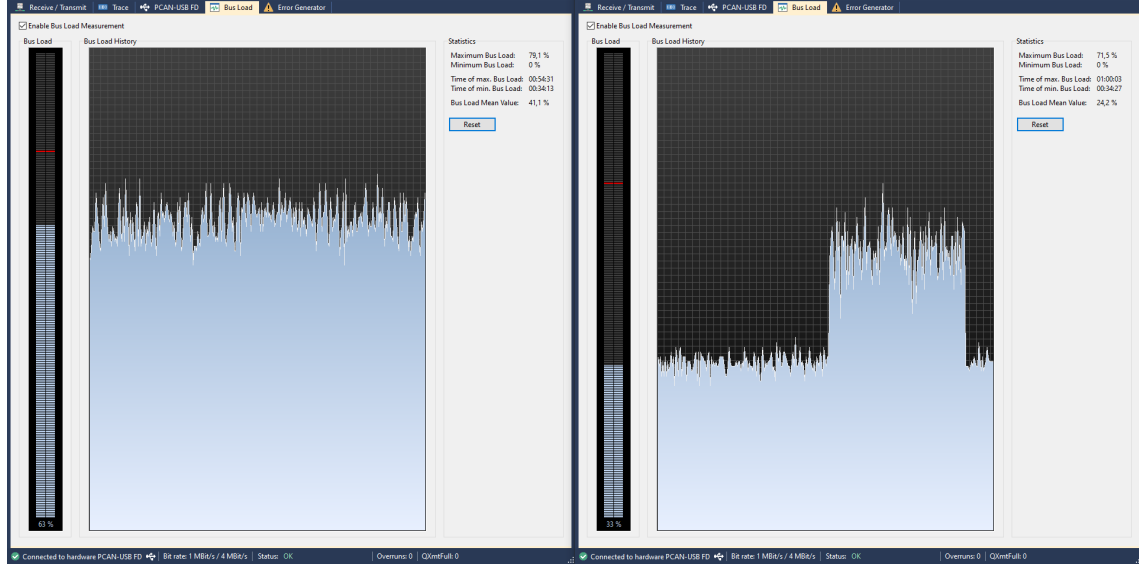


Figure 6: CAN-FD bus load in Nova during normal operation, screendumps from PCAN-View [10]. Vertical axis is utilization and horizontal axis is time. Utilization scale goes from 0% to 100%. Bus 0 on the left and bus 1 on the right.

From the traces we can see that bus 0 typically has a utilization of 60-70% while bus 1 rests at about 30% while out of DE mode and around 60% in DE mode. This is within the maximum utilization of 69%, calculated in section 2.2.2. Bus 0 is mostly within the limit, but has peaks as high as 79%, this means it is likely to have loss of packets.

we know that to ensure that no frames are dropped (i.e. all deadlines are met) the bus utilization should be no more than 69%. For bus 1, this seems to be in order. The trace tells us that the peak utilization is 71.3%, but this is an outlier and probably not an issue. However, bus 0 seems to have a somewhat high bus load continuously, staying around 70% load with peaks as high as 79.1%. This is an area that can be improved.

To start of, we will take a closer look at the frames that are transmitted on each bus. Last season, the team started the process of transitioning from an in-house protocol on top of CAN/CAN-FD (called the Revolve Protocol), to a standardized alternative, UAVCAN [15]. One of the benefits of using UAVCAN is a simplified message definition process. Messages are defined in Data Structure Definition Language (DSDL) files and can be converted to C for use in embedded systems and high level data formats like JSON for use in analytics software. As a part of this transition, a simple Python script for calculating the load on each bus based on message size and period was written by Åsmund Eek, a team member from last year that was responsible for the CAN/CAN-FD systems. The script is listed in the appendix. This script has been used during this project to calculate the worst-case bus utilization for the different embedded systems connected to the CAN-FD

buses.

Using the DSDL files to calculate the load on each CAN-FD bus on Nova, yields the results shown in tables 3 and 4.

Table 3: Frames on CAN-FD bus 0 in Nova. Common denotes frames that are sent from all or most systems.

Message origin	Utilization (%)
Common	0.01
AMS	15.4
CCC	0.01
SBS	54.0
<b>Total</b>	<b>69.4</b>

Table 4: Frames on CAN-FD bus 1 in Nova.

Message origin	Utilization (%)
VCU	36.1
Inverter	50.0
<b>Total</b>	<b>86.1</b>

Comparing the traces on the two buses from figure 6 to the calculated loads from tables 3 and 4, we can see that the calculated loads fits relatively well with the measured load. We observe that the estimated worst case load for bus 1 is somewhat higher than in the trace, this was later discovered to be a bug in the script which causes Revolve Protocol messages to be sent as extended frames, meaning all frames contains 18 extra bits and padding. All messages marked inverter in the table uses the Revolve Protocol. As this bug was discovered during the assembly and testing period, it was decided to not fix it. This project is about the VCU and its calculated bus load seems to be correct.

### 3.2.2 Ethernet for telemetry

It is necessary to retrieve data on the vehicle and the different systems on it during races. This is both for later analysis and to indicate to the team if there might be issues with the vehicle before things go wrong. To achieve this, Nova utilized a wireless communication solution from Radionor, the CRE2-144-LW [3]. UDP is used to interface with the radio and the physical layer is Ethernet. As no other system on Nova was equipped with Ethernet, the solution was to place a Raspberry Pi 3 B+ in the vehicle, and connect it to the CAN-FD buses with two PCAN-FD USB dongles. The Raspberry Pi would simply gather all available data from the CAN-FD buses and transmit it to the radio over Ethernet.

As the SoM chosen for VCU20 is equipped with an Ethernet-PHY interface, it is an opportunity to greatly simplify the telemetry system. By connecting the VCU directly to the radio using Ethernet, the total weight and complexity of the vehicle can be reduced. The downside to this is that the complexity is moved to the VCU, as it now has to

communicate with the radio in addition to everything else.

It is important that the telemetry system is able to perform at least as well as in Nova. As the previous telemetry system gathered data from both CAN-FD buses and sent the information over Ethernet to the radio, we must consider the bandwidth of both CAN-FD and Ethernet. The buses on Nova ran with a maximum data rate of 4MHz, although it does not use this data rate all the time, we can simplify and say it has a bandwidth of 4Mbps. This means a total bandwidth of both CAN-FD buses of 8Mbps. The Raspberry Pi 3 B+ has a maximum Ethernet bandwidth of 300Mbps [7]. This means the Ethernet interface on VCU20 has to achieve a minimum bandwidth of 8Mbps. According to the specifications, the Mercury ZX5 module is capable of Gigabit Ethernet (1Gbps) [5]. However, the actual speed of the interface is heavily dependent on the quality of the PCB layout. Gigabit Ethernet consists of four differential pairs, each of which should be matched in length to each other and within the pair to achieve maximum bandwidth.

An important note about the messages sent from the VCU is that they are not needed by other systems on the car, except for the telemetry as they are important for analysis. Moving the telemetry to the VCU should therefore reduce the load on the CAN-FD buses by as much as 36.1%, as seen in table 4. As the CAN-FD messages sent from the VCU.

Summing up, a system overview of VCU20 can be seen in figure 7.

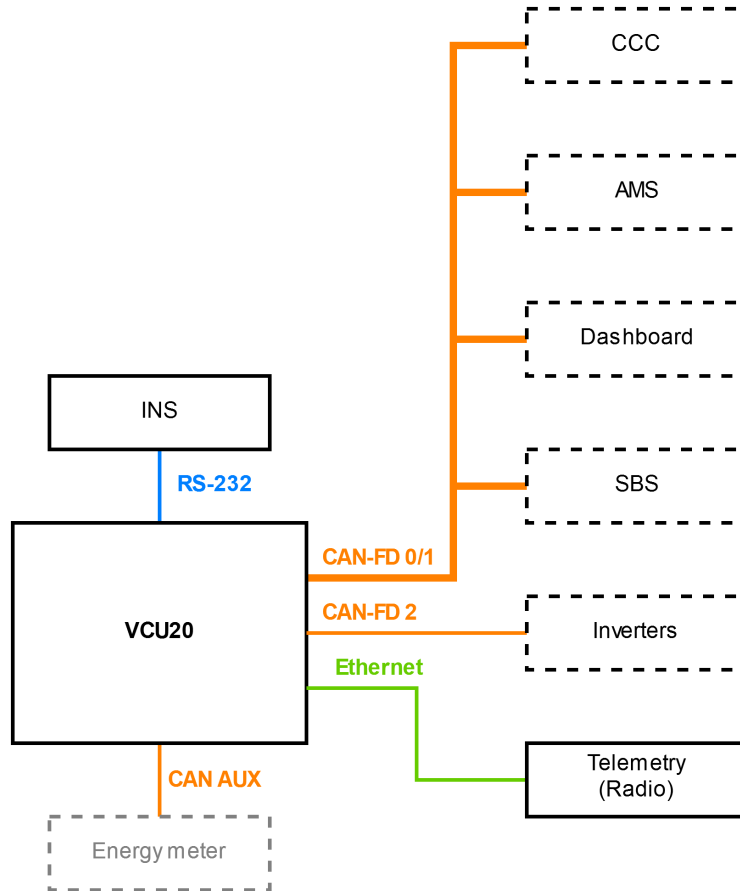


Figure 7: Author's VCU20 interface overview. Main focus on communication interfaces.



## 4 Results

### 4.1 Hardware design

The electronic design automation (*EDA*) software used for schematics and pcb layout, Altium Nexus [13] supports multiple features which shortened the design process significantly. Firstly, since VCU19 used *hierarchical design* (i.e. project is composed of multiple sheets and sub sheets), large parts of the schematics developed during last years design period was reused without issues.

Features for reuse in the PCB layout part of the design was also used. Altium Nexus has a feature called *rooms* which is a grouping of component footprints and how they are connected. This was used for the CAN-FD circuits.

The PCB was produced by Simpro AS as they are sponsoring Revolve NTNU. The finished, soldered PCB can be seen in figure 9.

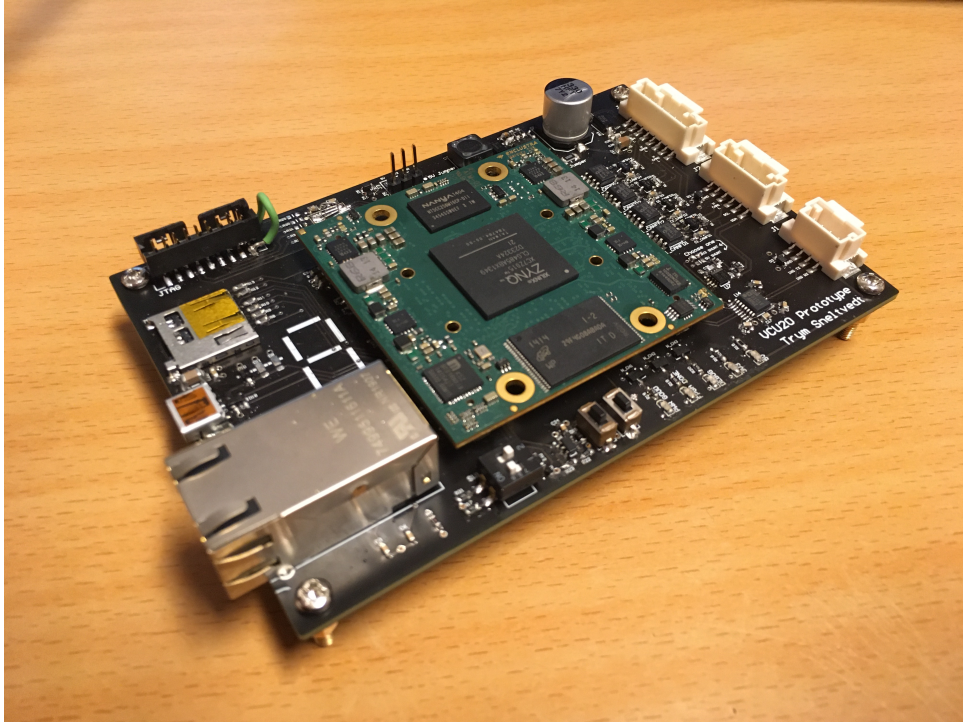


Figure 8: VCU20 PCB with all components and Mercury ZX5 module attached.

### 4.2 CAN-FD buses

Because of time shortage, a proper hardware-in-the-loop (HIL) test to determine the new load on the CAN-FD buses was not performed. It would have required porting the software for VCU19 to VCU20 and implement functionality for Ethernet. Because of the time frames set by the embedded electronics group in Revolve NTNU, the PCBs arrived in late November. In addition, the embedded electronics workshop had to be moved last minute, which also delayed the schedule.

However, the CAN-FD buses were tested and proved working. Although the bandwidth has yet to be tested, there is little to indicate any worse performance than that on VCU19.

### 4.3 Ethernet

Ethernet functionality was tested and verified. A simple web server part of Enclustra's example project for the ZX5 provided the source code and instructions [6], it worked perfectly. Unfortunately, there was no time to benchmark the bandwidth of the Ethernet interface and thereby determine the degree of success in regards to the differential pair layout.

### 4.4 Further work

The proposed solution of an extra CAN-FD bus solely used by the inverters and VCU has room for improvement. CAN-FD buses are designed for communication between several embedded systems. When there are only two embedded systems connected to the bus a point-to-point communication system might be more suitable, like the Ethernet link between the VCU and the telemetry radio. It would be best to utilize one of the unused interfaces on the module as this would reduce the required amount of supporting circuitry. The module is equipped with 4 general purpose Gigabit Transceivers, these would probably be a good fit.

Another solution that the author would recommend to next year's team is to consider merging the VCU and the inverter control. The VCU is currently overpowered when considering the relatively simple tasks it performs and it should at least be researched whether the Zynq-7000 could run the inverter control loops in addition to the torque vectoring algorithms. If the ZX5 module proves to be too weak, the team should consider transitioning to the Xilinx Zynq UltraScale+ MPSoC platform [11]. They are equipped with one quad/dual-core ARM Cortex A53 application processors, one dual-core ARM Cortex R5 real-time processor and a FPGA. This should be more than sufficient to run both systems. Possible pitfalls are competition rules, they must be examined closely. The Ultrascale platform is available as modules as well, although at a higher cost compared to the Zynq-7000 series.

## 5 Discussion

### 5.1 PCB design

The use of hierarchical design made the PCB design process relatively simple. Layout of the Ethernet interface required use of differential signalling, this is the most complex piece of hardware design in this project. Because of tools made specifically for this kind of work in the PCB layout software, this process is made easier even for inexperienced hardware designers.

## 5.2 PCB production

It is worth noting that the prototype production run of VCU20, a sponsor of Revolve NTNU covered all costs of PCB production. This renders the argument of reducing cost by employing a SoM somewhat moot. However, this has not been the case for previous teams, and it is far from certain that it will continue like this in the future.

## 5.3 PCB assembly

Soldering the finished PCB was done solely by hand without any major problems. The connectors used for the ZX5 module has a *pitch* (distance between the individual pins) of 0.5mm, which is hard to solder by eye. A microscope was used for this part of the soldering, and since Revolve NTNU owns several microscopes this is not regarded as a large issue.

## 5.4 PCB testing

After finishing assembly, VCU20 underwent a series of tests to ensure correct functioning. During testing, it was discovered that the Joint Test Action Group (JTAG) interface was incorrectly connected to the ZX5 module. See figure

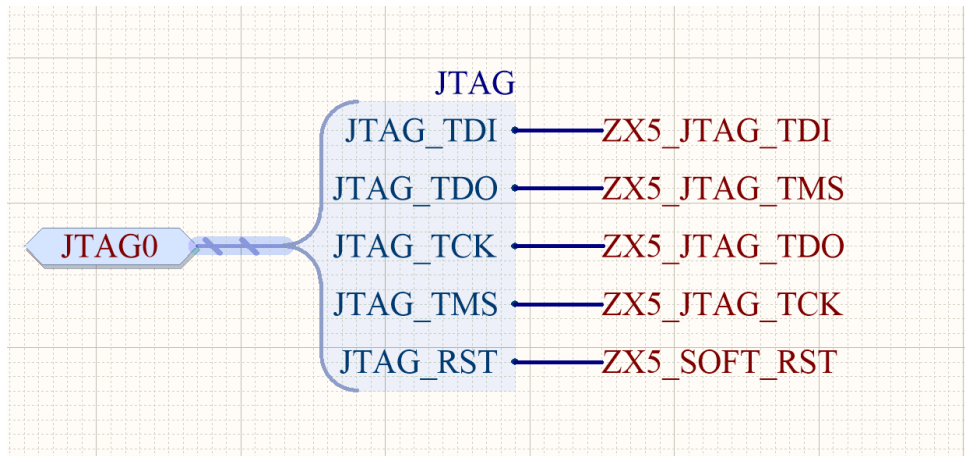


Figure 9: JTAG interface schematic in Altium Nexus. Notice the signal mismatch as they enter the *harness* (a collection of signals).

Luckily, this error was recoverable by unsoldering a programming buffer soldering jumper wires to its pads, correcting the error. After performing this fix, the JTAG worked correctly.

## 6 Conclusion

A Vehicle Control Unit built around an Enclustra Mercury ZX5 System-on-Module was successfully designed and tested. The hardware design was greatly simplified as a result of this, reducing design time. Also, by utilizing the Ethernet interface available on the module, vehicle weight was reduced by approximately 200g and load on CAN-FD buses can be reduced by as much as 36.1%.

Xilinx Zynq-7000 is a very advanced platform for running software, it is the view of the author that moving the system complexity from hardware to software is preferable to Revolve NTNU. This is both because most of the team members have more experience with software than hardware, and because the development cycle for software is much shorter than that of hardware.

## References

- [1] Alan Burns. *Real-time systems and programming languages : Ada, real-time Java and C/real-time POSIX*. eng. 4th ed. International computer science series. Harlow: Addison-Wesley, 2009, p. 371. ISBN: 9780321417459.
- [2] R. De Andrade et al.  
“Analytical and Experimental Performance Evaluations of CAN-FD Bus”.  
In: *IEEE Access* 6 (2018), pp. 21287–21295. ISSN: 2169-3536.  
DOI: 10.1109/ACCESS.2018.2826522.
- [3] Radionor Communications AS. *CRE-144-LW product information page*.  
URL: <https://radionor.no/product/cre2-144-lw/>. (accessed: 05.12.2019).
- [4] Enclustra. *Enclustra Mercury ZX5 product image*.  
URL: [https://www.enclustra.com/assets/images/products/soc\\_modules/mercury\\_zx5/mercury\\_zx5\\_perspective\\_web\\_new\\_1000.jpg](https://www.enclustra.com/assets/images/products/soc_modules/mercury_zx5/mercury_zx5_perspective_web_new_1000.jpg).  
(accessed: 18.12.2019).
- [5] Enclustra. *Mercury ZX5 product overview*. URL: <https://www.enclustra.com/en/products/system-on-chip-modules/mercury-zx5/>.  
(accessed: 11.12.2019).
- [6] Enclustra. *Mercury ZX5 Reference Design for Mercury PE1 V4*.  
URL: [https://download.enclustra.com/public\\_files/SoC\\_Modules/Mercury\\_ZX5/Mercury\\_ZX5\\_Reference\\_Design\\_for\\_Mercury\\_PE1\\_V4.zip](https://download.enclustra.com/public_files/SoC_Modules/Mercury_ZX5/Mercury_ZX5_Reference_Design_for_Mercury_PE1_V4.zip).  
(accessed: 19.12.2019).
- [7] Raspberry Pi fundation. *Raspberry Pi 3 B+ product overview*. URL:  
<https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/>.  
(accessed: 11.12.2019).
- [8] Formula Student Germany. *FSG rules 2020*. URL: [https://www.formulastudent.de/fileadmin/user\\_upload/all/2020/rules/FS-Rules\\_2020\\_V1.0.pdf](https://www.formulastudent.de/fileadmin/user_upload/all/2020/rules/FS-Rules_2020_V1.0.pdf). (accessed: 05.12.2019).
- [9] PEAK-System Technik GmbH. *PCAN-USB FD product overview*.  
URL: <https://www.peak-system.com/PCAN-USB-FD.365.0.html?&L=1>.  
(accessed: 18.12.2019).

- [10] PEAK-System Technik GmbH. *PCAN-View software overview*.  
URL: <https://www.peak-system.com/PCAN-View.242.0.html?&L=1>.  
(accessed: 17.12.2019).
- [11] Xilinx Inc. *Zynq Ultrascale+ product selection guide*.  
URL: <https://www.xilinx.com/support/documentation/selection-guides/zynq-ultrascale-plus-product-selection-guide.pdf>.  
(accessed: 04.12.2019).
- [12] Xilinx Inc. *Zynq-7000 product brief*.  
URL: <https://www.xilinx.com/support/documentation/product-briefs/zynq-7000-product-brief.pdf>. (accessed: 04.12.2019).
- [13] Altium Limited. *Altium Nexus product page*.  
URL: <https://www.altium.com/altium-nexus/>. (accessed: 19.12.2019).
- [14] PCBWay. *Prototype PCB online instant quote*.  
URL: <https://www.pcbway.com/orderonline.aspx>. (accessed: 9.12.2019).
- [15] UAVCAN development team. *UAVCAN website*. URL: <https://uavcan.org/>.  
(accessed: 17.12.2019).
- [16] *Xilinx CLG225 BGA package image*.  
URL: <https://media.digikey.com/Photos/Xilinx%20Photos/122;225CSBGA-1.5-13x13;CLG;225.JPG>. (accessed: 09.12.2019).

# Appendix

## Bus load calculation code

The following Python script was written by team member Åsmund Eek during the 2019 season. It reads the DSDL files from an (internal) centralized repository for keeping track of all message formats that are present on the CAN-FD buses. Note that the script differentiates between CAN frames using the UAVCAN data link layer protocol, and the deprecated Revolve Protocol (RP).

```
1 import argparse
2 import yaml
3 import os
4 import sys
5 import re
6 import math
7 import logging
8
9 logging.basicConfig(format='%(levelname)s:%(message)s', level=logging.INFO)
10 logger = logging.getLogger("bus_load_calc")
11
12 def get_all_subjects_from_namespace(source, namespace):
13
14     fixed_subject_id_pattern = re.compile(r"#define \w+_FIXED_SUBJECT_ID")
15
16     subjects = []
17
18     for root, _, files in os.walk(os.path.join(source, namespace), topdown=
19 True):
20         for file_ in files:
21             file_path = os.path.join(root, file_)
22             with open(file_path, "r") as f:
23                 if fixed_subject_id_pattern.search(f.read()):
24                     # Either service or message with non fixed subject id
25                     subjects.append(file_path)
26
27     return subjects
28
29 class Message:
30     def __init__(self, *args, **kwargs):
31         self._wc_bus_load = 0
32         self._bc_bus_load = 0
33         self._frames = []
34         self._wctts = []
35         self._bctts = []
36         self._frequency = -1
37         self._mtu = 64
38
39     @classmethod
40     def calculate_next_dlc(self, length):
41         CAN_FD_DLC = [0, 1, 2, 3, 4, 5, 6, 7, 8, 12, 16, 20, 24, 32, 48,
42 64]
43         for i in range(len(CAN_FD_DLC)):
44             if length <= CAN_FD_DLC[i]:
45                 return CAN_FD_DLC[i]
46
47     def _calculate_wctt(self, t_arb, t_data):
48         for p in self._frames:
49             wctt = 32 * t_arb + (28 + 5 * math.ceil((p - 16)/64) + 10*p) *
```

```

t_data
47         self._wctts.append(wctt)
48
49     def _calculate_bctt(self, t_arb, t_data):
50         for p in self._frames:
51             bctt = 29 * t_arb + (27 + 5* math.ceil((p - 16)/64) + 8*p) *
t_data
52             self._bctts.append(bctt)
53
54     def calculate_busload(self, t_arb, t_data):
55         self._calculate_wctt(t_arb, t_data)
56         self._calculate_bctt(t_arb, t_data)
57
58         for wctt in self._wctts:
59             self._wc_bus_load += wctt * self._frequency * 100
60
61         for bctt in self._bctts:
62             self._bc_bus_load += bctt * self._frequency * 100
63
64     def calculate_payload_per_frame(self):
65         pass
66
67 class UavcanMessage(Message):
68     def __init__(self, path):
69         super().__init__()
70         self._path = path
71         self._name = self._get_subject_name()
72         self._payload = self._get_max_payload_length()
73         self._frames = []
74         self._wctts = []
75         self._frequency = -1
76         self._bus_load = 0
77
78     def _get_subject_name(self):
79         """ Return the full name for a subject"""
80         subject_name_pattern = re.compile(r"#define \w+_NAME +\"([a-zA-Z0-9
81         _.]*)\"")
82         with open(self._path, "r") as f:
83             return subject_name_pattern.findall(f.read())[0]
84
85     def _get_max_payload_length(self):
86         """ Return the number of Bytes of payload for a given subject"""
87         subject_name_pattern = re.compile(r"#define \w+_MAX_SIZE
88         +([0-9\\(\\)\\+\\[/ ]*)")
89         with open(self._path, "r") as f:
90             return int(eval(subject_name_pattern.findall(f.read())[0]))
91
92     def calculate_payload_per_frame(self):
93         """
94         Return a list containing the payload of each frame needed to
95         transmit
96         the user payload.
97         This includes:
98             - 1 Byte UAVCAN tailbyte per CAN(-FD) frame
99             - 2 Byte CRC for multiframe UAVCAN messages
100             - Padding bytes to get to closest DLC for CAN-FD
101         """
102         payloads = []
103         current_payload = 0
104         user_payload = self._payload

```

```

103     # Use a greedy approach to fill one and one frame
104     while user_payload > 0:
105
106         if user_payload > (self._mtu - 1):
107             current_payload += (self._mtu - 1)
108         else:
109             current_payload += user_payload
110
111         # Decrement the total number of Bytes left of user payload
112         user_payload -= current_payload
113
114         # Add tailbyte
115         current_payload += 1
116
117         if user_payload == 0 and len(payloads) == 0: # Singleframe
118
119             # Need to pad until next valid DLC
120             payloads.append(self.calculate_next_dlc(current_payload))
121         elif user_payload == 0 and len(payloads) > 0: # Multiframe
122             if current_payload < (self._mtu - 1): # The two CRC bytes
fits into the last multiframe
123
124                 current_payload += 2
125                 payloads.append(self.calculate_next_dlc(current_payload
126             ))
127             elif current_payload == (self._mtu - 1): # The CRC must be
split over two frames
128
129                 # Append one of the two CRC bytes to the current frame
current_payload += 1
130
131                 # Append the last full frame
132                 payloads.append(self.calculate_next_dlc(current_payload
133             ))
134
135                 # Append the last frame containing 1 Byte CRC and
tailbyte
136                 payloads.append(2)
137             else: # Both CRC bytes must be put in new frame
138
139                 payloads.append(self.calculate_next_dlc(current_payload
140             ))
141
142                 # Append the last frame containing 2 Byte CRC and
tailbyte
143                 payloads.append(3)
144             else: # Middle of multiframe
145                 payloads.append(self.calculate_next_dlc(current_payload))
146
147             # Reset payload for next frame
current_payload = 0
148
149     self._frames = payloads
150
151     class RevolveProtocolMessage(Message):
152     def __init__(self, name, frequency=None, size=None):
153         super().__init__()
154         self._name = name
155         self._frequency = frequency
156         self._size = size

```



```

157     def calculate_payload_per_frame(self):
158         assert self._size <= self._mtu
159
160         self._frames.append(self.calculate_next_dlc(self._size))
161
162     def update_frequencies(messages, default_frequency,
163                             user_registered_frequencies):
164         subjects = dict((message._name, message) for message in messages)
165         # print(subjects.values())
166         for subject, freq in user_registered_frequencies.items():
167             if subject not in subjects:
168                 logger.error(f"Got unknown subject when registering frequencies
169                             : {subject}")
170                 sys.exit(1)
171
172                 subjects[subject]._frequency = freq
173
174             # Log a warning if a frequency is not set
175             for name, subject in subjects.items():
176                 if subject._frequency == -1:
177                     logger.warning(f"No frequency was found for '{subject._name}',
178                                     is set to default value '{default_frequency}'")
179                     subjects[name]._frequency = default_frequency
180
181             return list(subjects.values())
182
183
184     def main():
185         parser = argparse.ArgumentParser()
186
187         parser.add_argument("-c",
188                             "--config",
189                             dest="config_path",
190                             required=True,
191                             help="Path to config file")
192
193         args = parser.parse_args()
194
195         config_file_path = os.path.abspath(args.config_path)
196
197         with open(config_file_path, "r") as f:
198             config = yaml.safe_load(f)
199
200         # MTU size
201         mtu = config["general"]["MTU"]
202
203         # Default frequency for signals
204         default_freq = config["general"]["default_frequency"]
205
206         # Get the time periods for both arbitration and data phase
207         t_arb = 1/float(config["general"]["arbitration_bit_rate"])
208         t_data = 1/float(config["general"]["data_bit_rate"])
209
210         # Get source path of generated C files
211         source = os.path.abspath(os.path.join(os.path.dirname(config_file_path),
212                                                 config["uavcan"]["source"]))
213
214         # Extract a flattened list of all periodic UAVCAN subjects

```

```

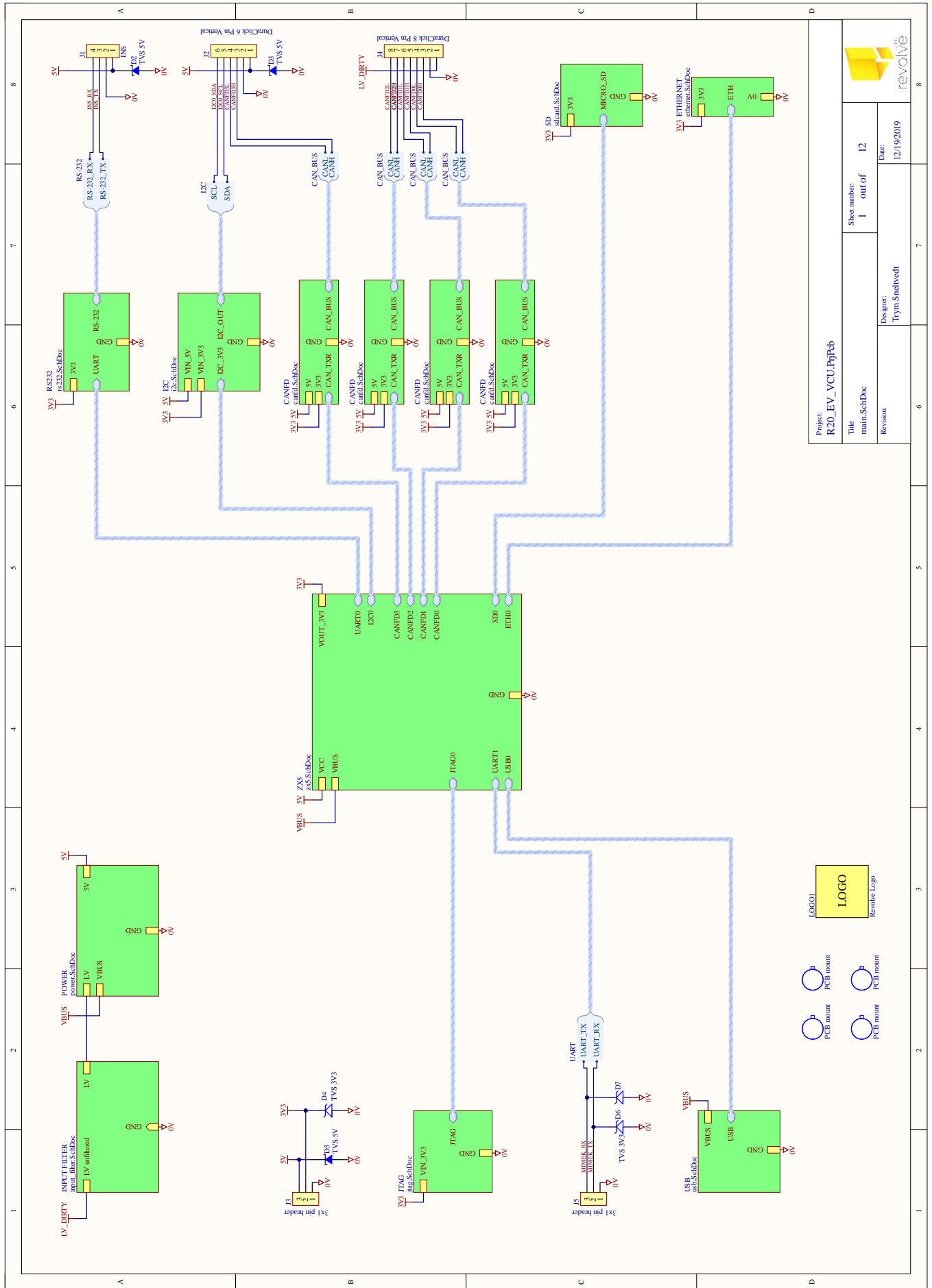
214     subjects_paths = []
215     for namespace in config["uavcan"]["namespaces"]:
216         subjects_paths.extend(get_all_subjects_from_namespace(source,
217             namespace))
218
219     subjects = []
220     for path in subjects_paths:
221         subjects.append(UavcanMessage(path))
222
223     subjects = update_frequencies(subjects, default_freq, config["uavcan"][
224         "frequencies"])
225     # RP messages
226     for message, value in config["RevolveProtocol"].items():
227         subjects.extend([RevolveProtocolMessage(message + "_" + str(i),
228             frequency=value["frequency"], size=value["size"]) for i in range(value[
229                 "amount"])]])
230
231     logger.info("Found the following messages")
232
233     worst_case_bus_load = 0
234     best_case_bus_load = 0
235
236     longest_subject_name = len(max(subjects, key=lambda x: len(x._name)).
237         _name)
238
239     for subject in subjects:
240         subject.calculate_payload_per_frame()
241         subject.calculate_busload(t_arb, t_data)
242         worst_case_bus_load += subject._wc_bus_load
243         best_case_bus_load += subject._bc_bus_load
244         logger.info(f"\t - {subject._name:<{longest_subject_name}}:
245             Frequency: {subject._frequency:>3}, Worst Case Busload: {subject.
246                 _wc_bus_load:>5.2e}, Best Case Busload: {subject._bc_bus_load:>5.2e}")
247
248     logger.info("")
249
250     logger.info(f"Total worst case bus load: {worst_case_bus_load:.2f}")
251     logger.info(f"Total best case bus load: {best_case_bus_load:.2f}")
252
253
254
255 if __name__ == "__main__":
256     main()

```

## PCB schematics and layout

The layout pages (last 4) are listed in the following order:

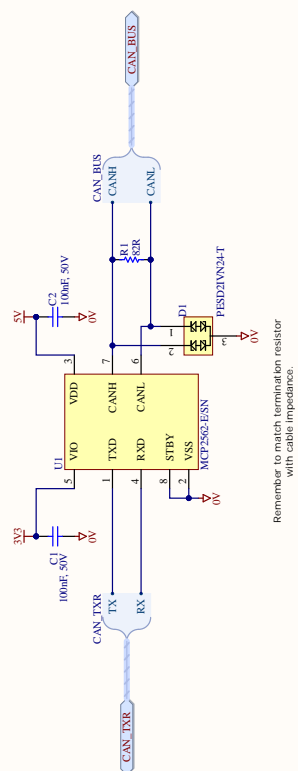
1. Top signal plane (red)
2. Ground plane (dark green)
3. Power plane (light green)
4. Bottom signal plane (blue)



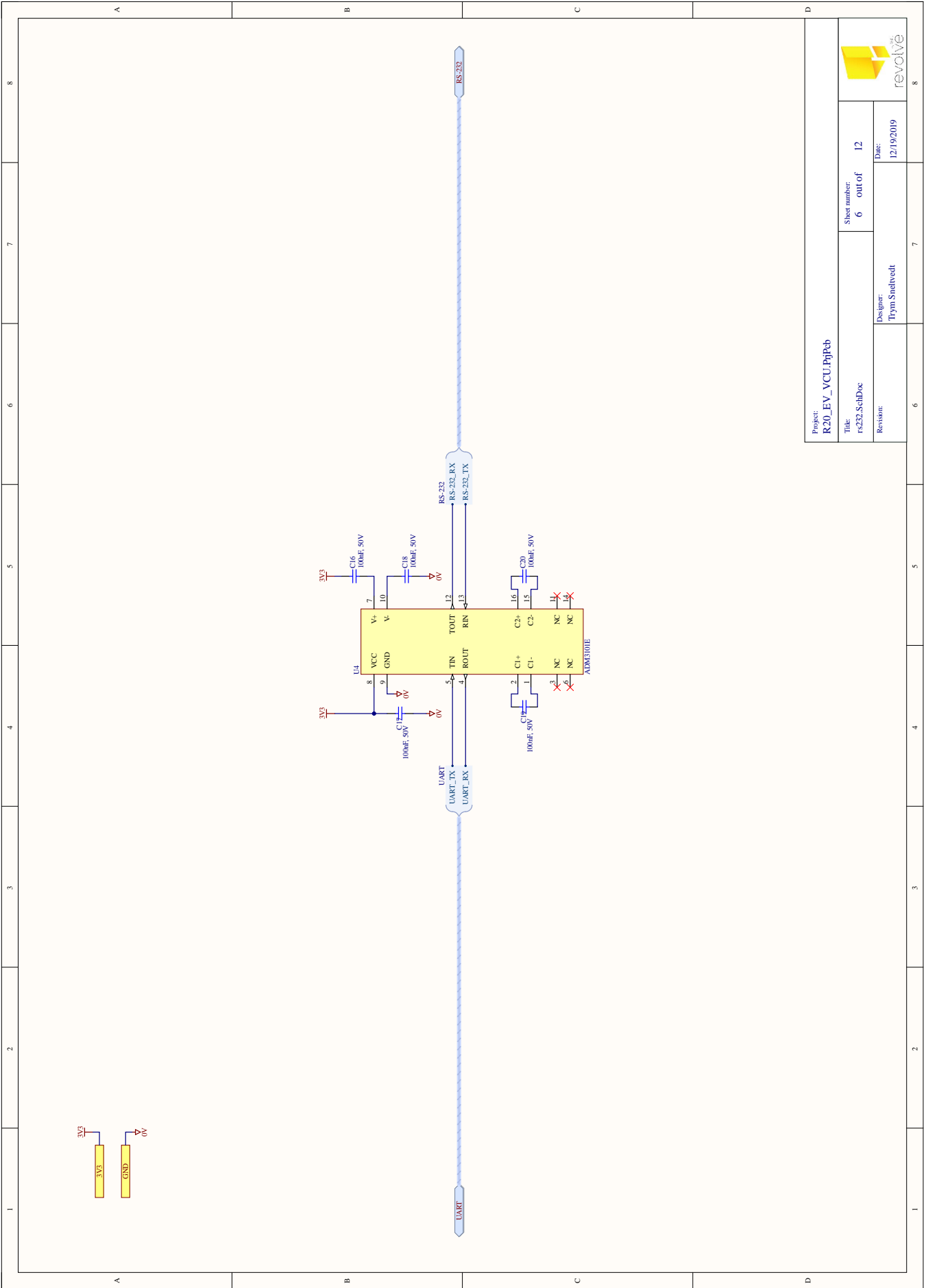


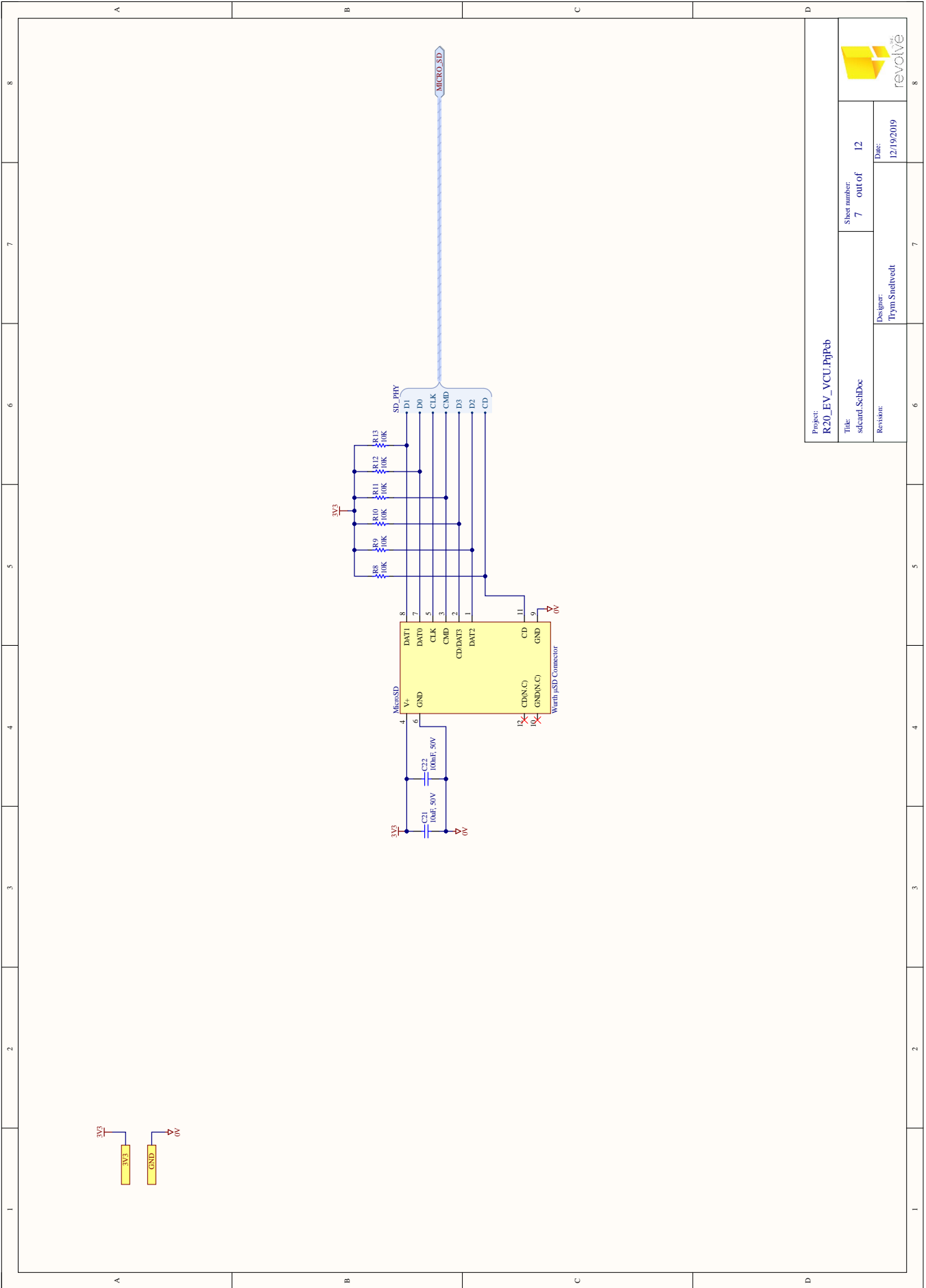









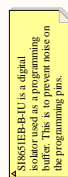
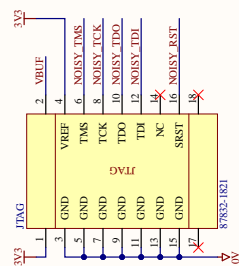


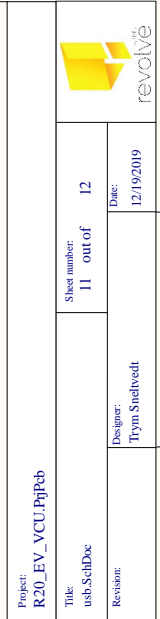


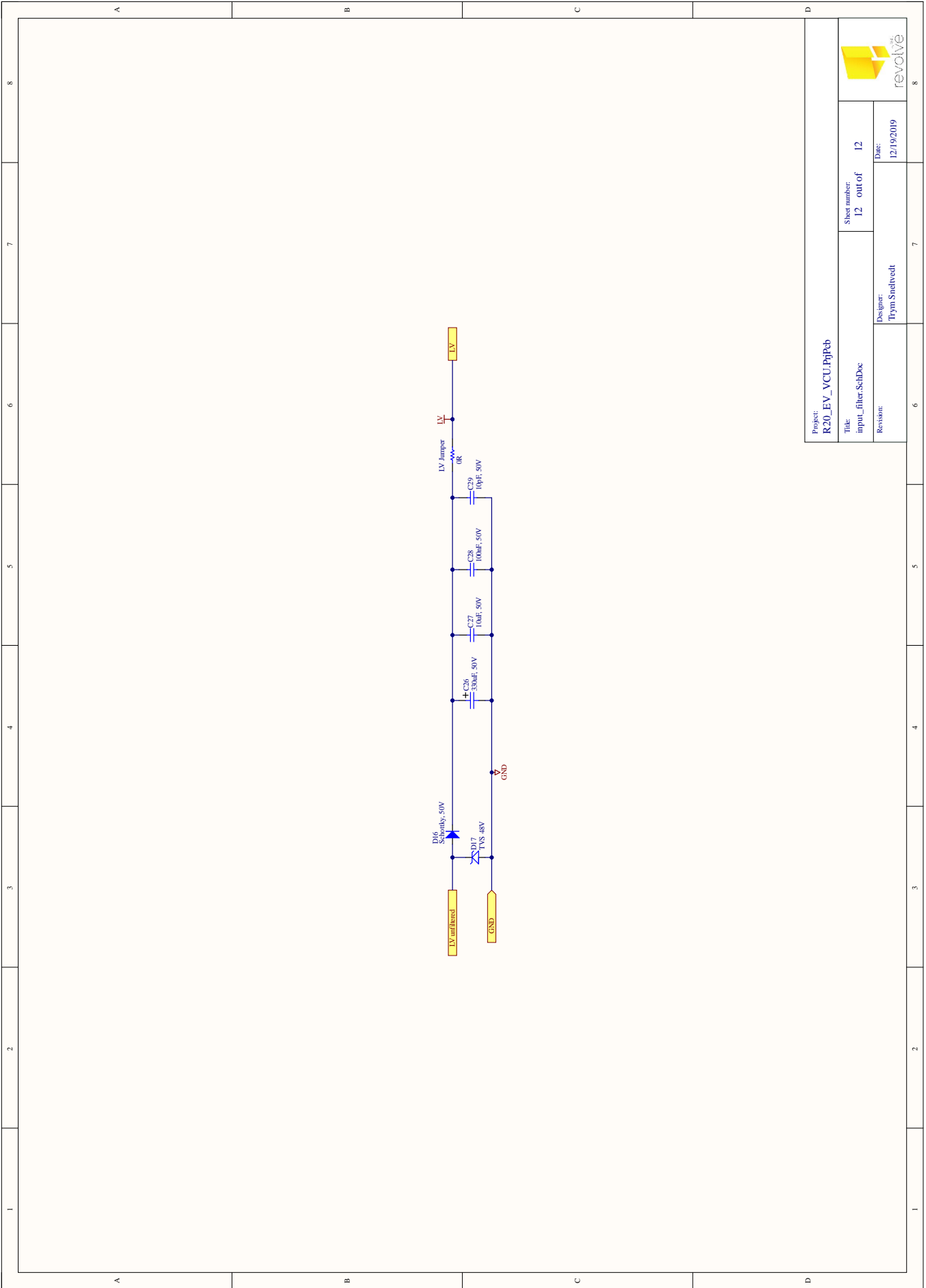
Project: R20_EV_VCU.PjFeb		Sheet number: 7 out of 12	
Title: sdcard.SchDoc	Designer: Tym Shelvedt	Date: 12/19/2019	
Revision:			












Project: R20_EV_VCU.PjPcb		
Title: input_filter.SchDoc	Sheet number: 12 out of 12	
Revision:	Designer: Tym Sheldt	
	Date: 12/19/2019	

