4-14-2020

Sunseeker Display and Driver Controller

Alec Kwapis
Western Michigan University, aleckwapis@gmail.com

Sunseeker Display and Driver Controller

Alec Kwapis, Conner McCarthy, and Nathan Heffington

Dr. Bradley Bazuin

ECE 4820 ELECTRICAL & COMPUTER ENGINEERING DESIGN II

April 14, 2020

Abstract

Digital dashboard displays with critical driver information are found in all modern vehicles. Examples of such information available to the driver include a speedometer, odometer, engine RPM, fuel gauge and more. The current 2016 Sunseeker solar car already has numerous displays that can show critical information to the driver, however, there are several problems that exist. Each display itself is less than two inches in size, the text on the screens is difficult to read, and the measurements have no units. Furthermore, these displays were made by a company that no longer exists, thus preventing the solar car team from purchasing any more for their 2020 version of the vehicle. Therefore, the next version of the Sunseeker solar car required a next generation driver display that supports strict power, weight, and space considerations. A 7-inch full-color LCD touch screen display with a Serial Peripheral Interface (SPI) was chosen as the new digital dashboard of the solar car. This display was integrated into the Driver Control unit of the existing solar car, which already possessed some of the critical driver information. The new Display and Driver Control (DDC) unit integrates in a single subsystem driver switches, controls, accelerator measurements, and CAN bus communications with interfacing for the display. CAN bus data and software programming allow for a screen of customized vehicle information to be available to the driver. A custom printed circuit board and housing has been designed, developed and tested, resulting in a DDC with fewer electronic modules, less wires and cables, and significantly greater capability than before.

DISCLAIMER

This report was generated by a group of engineering seniors at Western Michigan University. It is primarily a record of a project conducted by these students as part of curriculum requirements for being awarded an engineering degree. Western Michigan University makes no representation that the material contained in this report is error free or complete in all respects. Therefore, Western Michigan University, its faculty, its administration or the students make no recommendation for use of said material and take no responsibility for such usage. Thus persons or organizations who choose to use said material for such usage do so at their own risk.

WESTERN MICHIGAN UNIVERSITY
COLLEGE OF ENGINEERING AND APPLIED SCIENCES
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
KALAMAZOO, MICHIGAN 49008

SENIOR DESIGN PROJECT REPORT RELEASE FORM

In accordance with the "Policy on Patents and Release of Reports Resulting from Senior Design Projects" as adopted by the Executive Committee Of the College of Engineering and Applied Sciences on Feb. 9, 1989, permission is hereby granted by the individuals listed below to release copies of the final report written for the Senior Design Project entitled:

PROJECT TITLE: Sunseeker Display and Driver Controller

PROJECT SPONSOR* **Did this project have a sponsor? YES ✓ (see footnote) NO_____**

Contact person and email address &/or telephone: Dr. Bradley Bazuin (brad.bazuin@wmich.edu)

Company Name: WMU Sunseeker Solar Car team

Design team has requested sponsor to verify in writing to course coordinator that all promised deliverables have been received. **YES____ NO✓** (please check)

TEAM MEMBERS NAMES:

NAME PRINTED	NAME SIGNED	DATE
Alec Kwapis	Alec Kwapis	4/13/20
Conner McCarthy	Conner McCarthy	4/13/20
Nathan Heffington	Nathan Heffington	4/13/20
_____	_____	_____
_____	_____	_____

* Those teams with a sponsor must have sponsor provide the **course coordinator** with written evidence that they have provided the sponsor with a copy of the final project report as well as with other items that the team has promised to the sponsor. The evidence could be a short note via email, fax or US mail from the sponsor indicating receipt of a copy of the report and all promised deliverables.

Acknowledgements/Permissions

Thank you to Dr. Bradley Bazuin, advisor of the project, and the Sunseeker solar car team for providing their time, funds, resources, and laboratory space throughout the course of the project.

Special thank you to Austin Gilbert and Velocity Design & Engineering, for 3D printing the housing for the printed circuit board.

We allow and encourage solar car teams to reproduce and improve on this project for their own solar cars.

Table of Contents

List of Tables

List of Figures

Introduction

Background

The Western Michigan University Sunseeker solar car team has been designing, building, and racing solar powered vehicles in the American Solar Challenge for nearly 30 years. The vehicle contains a large solar array that covers nearly the entire topside of the vehicle, which is meant to capture as much energy from the sun as possible. The solar array charges the main battery, which provides power to the vehicle's electric motors and the rest of its subsystems. In the cockpit of the vehicle sits the driver's controls and the displays which show the driver critical information, such as speed, current battery voltage, maximum and minimum cell voltages, and more. However, these displays have proved to be a problem for the Sunseeker team in several ways.

The screen of the displays are less than two inches in size, which only allows one value to be viewed at a time. The driver must switch between values by pressing a button located near the displays. Drivers of the solar car have reported that the screens are difficult to read, which can be attributed to the viewing angles and brightness of the panels. Furthermore, values displayed on the screens have no units because there is not enough room on each screen to display both the value and the unit label. Each display has a set of LEDs that show which value is currently being displayed. These LEDs are also very hard to see in the daylight. Perhaps the largest issue with these displays is that they can no longer be purchased, as their manufacturer, Tritium, has gone out of business. In fact, this is also true for the existing driver controller as well.

The driver controller module connects directly to the driver's controls in the cockpit and controls switches such as the turn signals, ignition, forward or reverse, the brake, and perhaps the most important – the accelerator pedal. The existing driver controller came in a discrete box that

could be purchased from Tritium and added to any solar car with other Tritium modules, such as the displays.

Problem Statement

Because of the problems with the current displays, and the fact that the original manufacturer no longer exists, a new display was added to the 2020 version of the solar car and combined with the recreation of the existing driver controller module. The driver controller is already responsible for setting the speed of the vehicle, which is a critical measurement that is displayed on the new screen. Because of this, it was logical to combine the display and driver controller module into a single module, which is now called the Display and Driver Control (DDC) unit. More critical measurements were added to the new display, such as battery voltage, minimum and maximum cell voltages, maximum cell temperature, state of charge, and net current of the battery. The new 7-inch full-color LCD touchscreen display has a user interface that is easy to read and understand. Furthermore, adding new measurements to the display is made easy by the well-structured and modular code that was written for it. A custom printed circuit board (PCB) was designed for the recreation of the driver controller, which connects to the display via a ribbon cable. 3D printed housings for both the PCB and the display were designed to prevent damage and make mounting in the solar car easier.

Discussion

Hardware Design

Display

The electronic hardware components consist of a PCB and the display. The display itself was purchased and consists of a 7-inch 800x480 full-color TFT resistive touch LCD panel, and an FT812 display controller. The display is pictured below in Fig. 1.

Figure 1: NHD-7.0-800480FT-CSXV-T

Although the display is touchscreen, this feature was not used nor was it prototyped during any part of the project. However, all the connections made during the PCB design phase support the touch screen interface. The software is the only part that would need to be changed to support the touchscreen. The display's panel features a software-controlled backlight, which allows it to be adjusted from completely off and up to 100% brightness. Because the display uses around 3W of power at 100% backlight brightness, the backlight was configured at 15% brightness, which still

allows the content on the screen to be viewed easily. The FT812 display controller is the interface between the actual panel and microprocessor of the DDC's PCB.

Block Diagram

A high-level block diagram of the PCB and the display is shown below in Fig. 2.

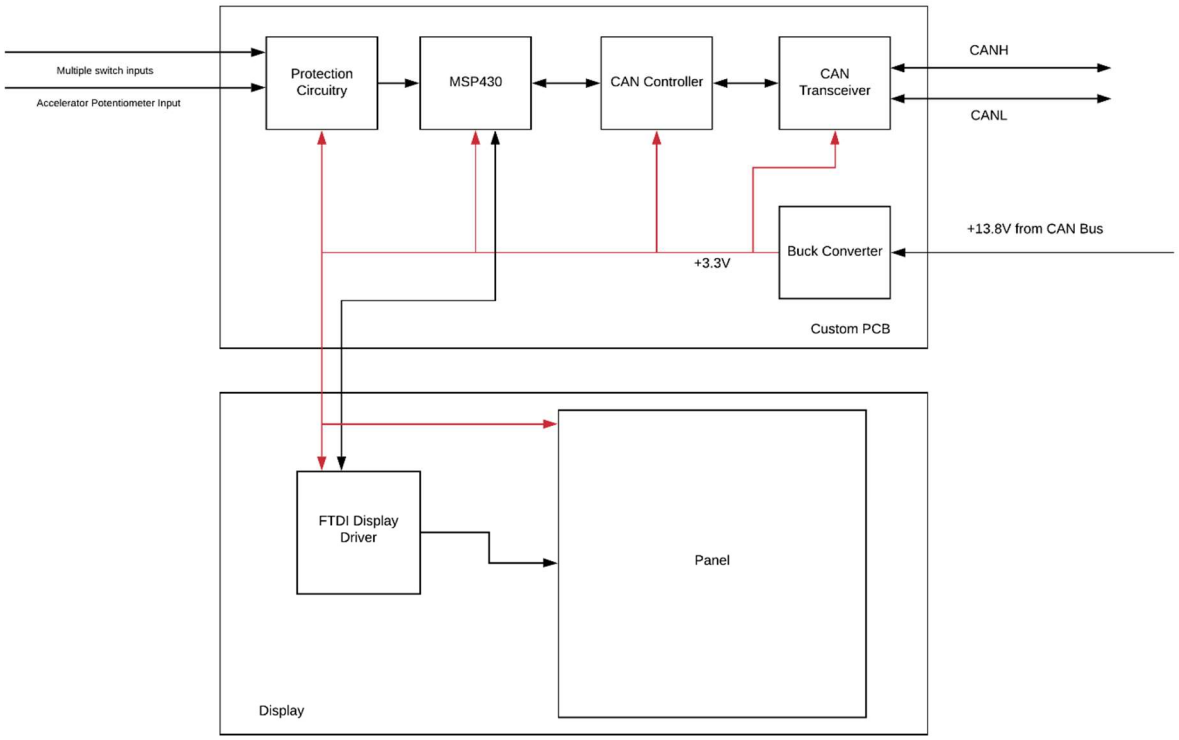


Figure 2: High-level block diagram

The display is depicted in the bottom block. The DDC's PCB is shown in the upper block. Within the display, the FT812 (labeled as 'FTDI Display Driver') connects directly to the panel. The only connections that the DDC PCB makes with the display is for power (the red line) and for communications (the black line). Both the display controller and panel are powered by 3.3V, which comes from the DDC PCB. The communication interface between the DDC PCB and the display controller is the Serial Peripheral Interface (SPI). At the center of the DDC PCB (labeled as 'Custom PCB') is the MSP430F5438A, which is a specific variant of the MSP430 family of

microprocessors manufactured by Texas Instruments that has been used in the Sunseeker solar car in other modules. It is for this reason, and the fact that the MSP430 is extremely low power, that this variant was selected for use in this project.

The driver control inputs (labeled 'Multiple switch inputs') and the accelerator potentiometer input both connect to the protection circuitry block. Because the driver's switches are powered from a 12V rail, each input must have its own set of protection circuitry so that the input pins of the MSP430 are not damaged by the high voltage. The CAN controller is an integrated circuit (IC) that processes the logic for the CAN bus. It consists of transmit and receives buffers that store CAN messages to be sent to and received from the CAN bus. The MSP430 communicates with the CAN controller via an SPI interface. The CAN controller communicates to the CAN transceiver via a set of TX and RX signals. The CAN transceiver is responsible for converting the logic high and logic low level signals from the CAN controller into a pair of differential signals titled CANH (CAN high) and CANL (CAN low). These are the standard signals of any CAN bus, and they connect to all the other modules in the solar car. The Sunseeker's CAN bus also contains power for every module connected to it, which comes in the form of 13.8V. Therefore, the DDC PCB contains a buck converter that steps down the 13.8V to 3.3V, which is used by every IC on the board.

Electrical Schematics

The schematic and PCB layout for the DDC was created in Altium Designer. The electrical schematic of the DDC PCB is shown on the following pages in Fig. 3, Fig. 4, and Fig. 5.

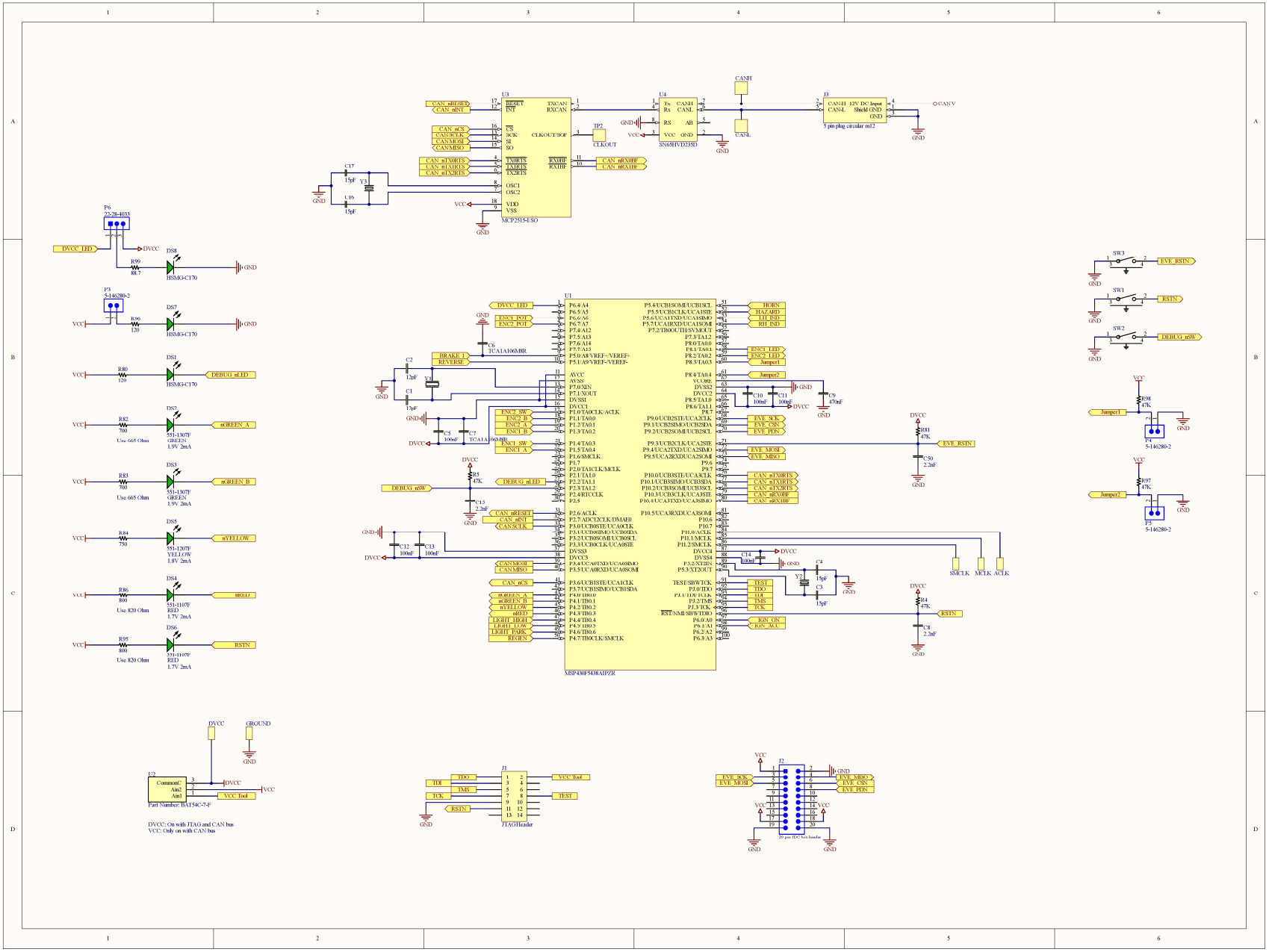


Figure 3: Electrical Schematic Sheet 1

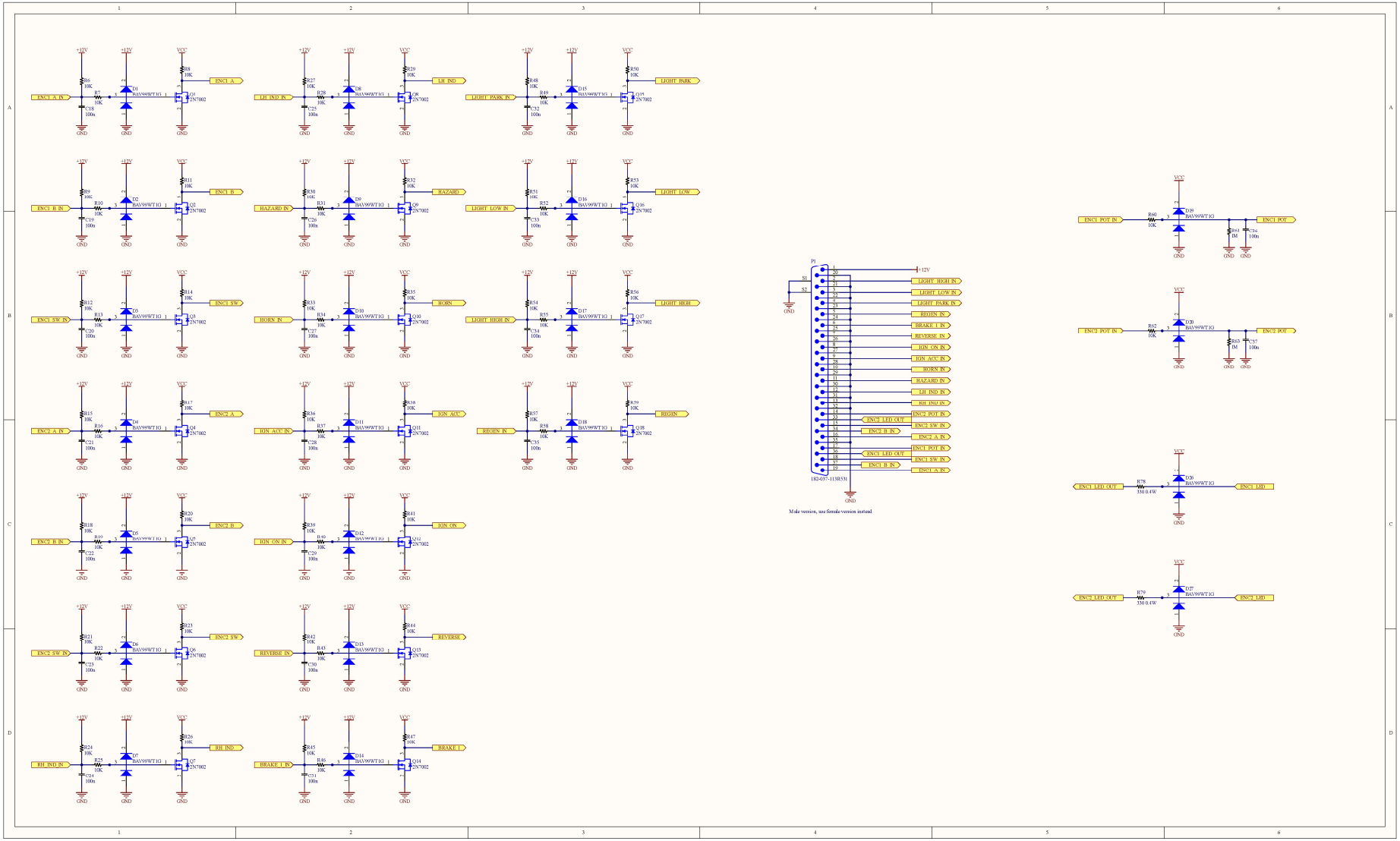


Figure 4: Electrical Schematic Sheet 2

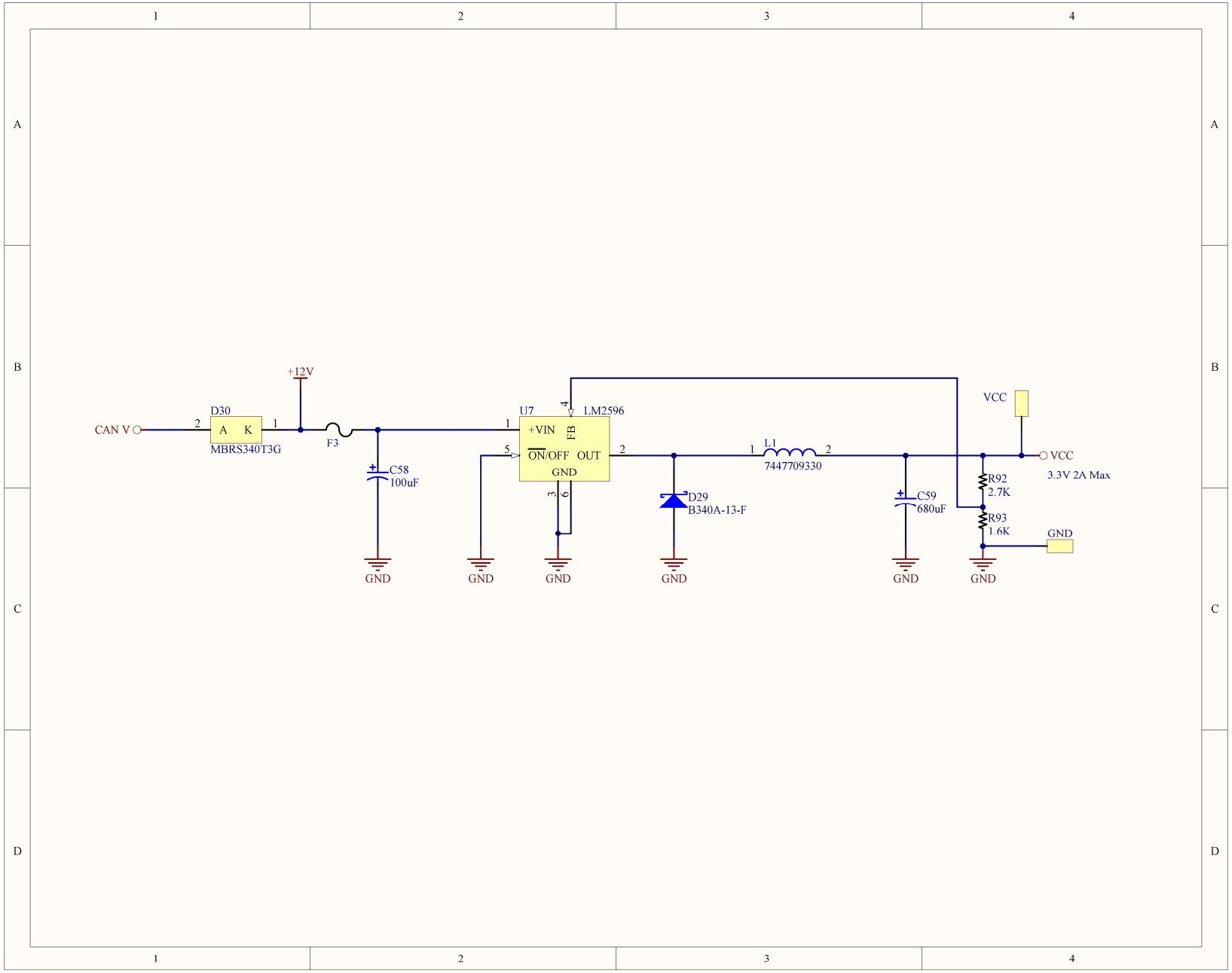


Figure 5: Electrical Schematic Sheet 3

Fig. 3 contains the main schematic for the DDC PCB. The MSP430F5438A is depicted in the center, with external crystals and bypass capacitors. While the external crystals are not necessary for the function of the MSP430, they generate a more precise clock over using the built in digitally controlled oscillator (DCO). The bypass capacitors are necessary for the function of the MSP430. At the top of the schematic is the MCP2515 (CAN controller), which connects to the SN65HVD235D (CAN transceiver). On the left side of the schematic are LEDs that are used for debugging and showing the status of the driver controller. On the right side of the schematic are buttons and jumpers, all of which (except the reset button) can be controlled via software to perform a specific function. The bottom of the schematic contains two connectors. The first connector is a JTAG header, which is used by a debugger such as the MSP-FET to reprogram the MSP430 with a new version of its software. The second connector is the header for the display, which is a 20-pin IDE box header. The leftmost part (BAT54C) contains a series of diodes that allows only the MSP430 to be powered and reprogrammed using the MSP-FET. When the CAN bus cable is connected, the CAN bus powers all the components on the board. However, when the CAN bus cable is disconnected and the debugger is connected via the JTAG header, only the MSP430 is powered so that it can be reprogrammed and debugged. Both the CAN bus cable and the debugger can be connected at the same time; however, it is not recommended.

Fig. 4 contains the schematic for the protection circuitry. There are 18 digital inputs to the driver controller, 2 analog inputs, and 2 digital outputs that are currently configured as timer modules. The DB37 connector is the same connector used in the existing driver controller, therefore, it was logical to choose the same connector and place the input and output pins in the same place and orientation so the cable in the solar car would not have to be modified.

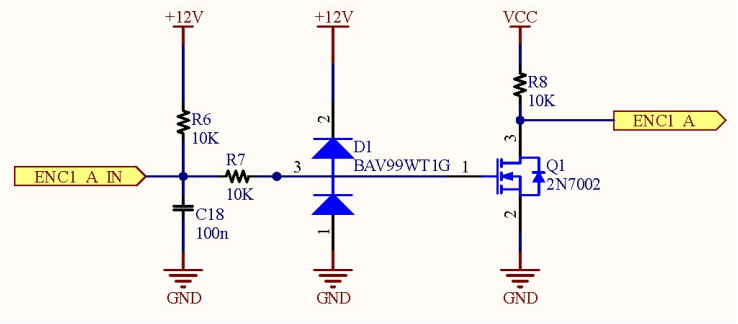


Figure 6: Protection Circuitry for a Digital Input

The same protection circuitry existed on the previous driver controller, and therefore was recreated for the new DDC. Fig. 6 shows one of these circuits for one of the rotary encoders in the solar car. All digital inputs work in the same manner, and so the one in Fig. 6 will be used as an example. The ENC1_A_IN input comes from the DB37 connector, which is connected to a rotary encoder's 'A' pin. The input pin of any digital input circuit can be connected to any single-pole-single-throw (SPST) switch, pushbutton, or a more complex switch as long as there is a logic high and logic low value provided by the switch. If a normally open pushbutton switch was connected to ENC1_A_IN, then its other pin should be connected to ground. When the button is open, the MOSFET sees 12V at its gate, which turns on the 2N7002, thus pulling ENC1_A to 0V. If the button is being pressed, the MOSFET sees 0V at its gate, thus pulling ENC1_A to VCC, which is 3.3V. This is how the digital inputs function on a 12V rail, but the MSP430 only ever sees 0V or 3.3V, thus protecting the MSP430 from the high voltage.

Fig. 5 contains the schematic for the DDC's power supply. The CAN V input on the left is connected directly to the 13.8V from the CAN bus. The VCC output on the right supplies the board's components with 3.3V. This is a custom design for a buck converter using an LM2596, which is a step-down voltage regulator. This means that the CAN bus voltage can vary significantly, and the LM2596 will always try to keep VCC at 3.3V. The values of R92 and R93

determine the output voltage and the feedback (FB) pin of the LM2596 is what causes it to maintain the current output voltage. The diode D30 and the fuse F3 are used as protection measures for the CAN bus, on top of creating the 12V rail. In fact, 12V is merely used as a naming convention, this voltage is closer to 13.3V because of the voltage drop across the Schottky diode. Nonetheless, this voltage difference does not impact the switches that are connected to its rail. With the display at 15% backlight brightness, the power supply generates an efficiency of around 60%. Furthermore, the DDC draws around 100 mA of current at 13.8V, to produce a power draw of less than 1.5W.

PCB Layout

Once the schematic for the DDC was finished, the PCB layout was started. Creating the PCB layout was done in Altium designer, which is made easy by simply adding a PCB to the project and importing the schematic to the PCB. The layout editor will generate ratsnest lines, which logically shows the connections that must be made on the PCB by using traces. The completed layout for the PCB is shown below in Fig. 7. The board itself was designed to be 9.5” by 6.5”, with four screw holes sized for M5 screws. The stackup of the PCB consists of only two layers, a bottom layer (shown in blue) and a top layer (shown in red). Ground planes were added to both layers, with stitching vias placed between the layers to create a strong ground connection. The right side of the board consists of the protection circuitry and the DB37 connector. The MSP430 is placed toward the center of the board with its bypass capacitors and crystals placed close to the chip. The rest of the connectors were placed on the left-hand side of the board with the LEDs.

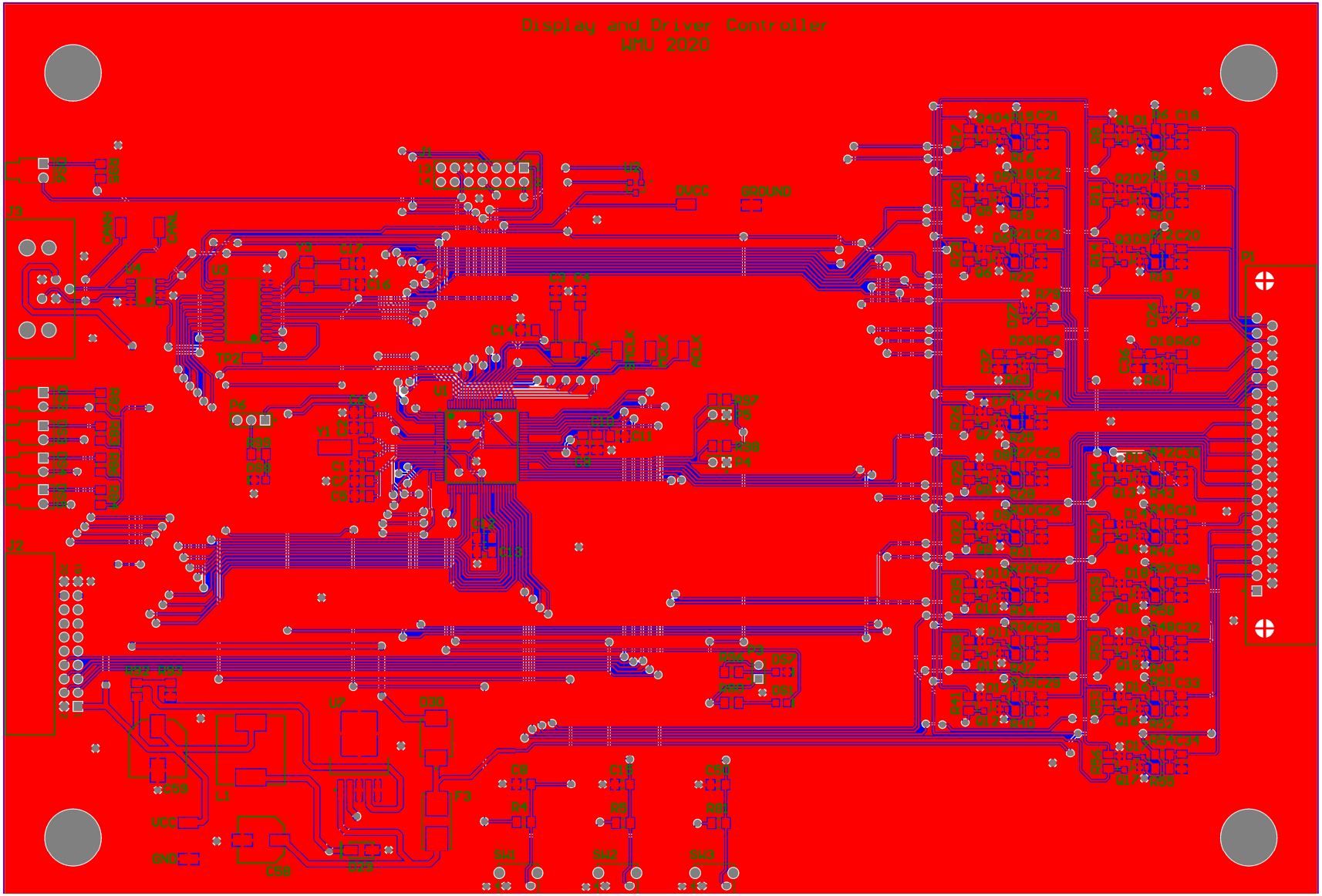


Figure 7: PCB Layout

The trace thickness was kept at ten mils (ten thousandths of an inch) except in the power supply, where the trace thickness was set to 35 mils. The thicker traces in the power supply were used so that the high current draw of the display would drop less voltage across the traces. This was also the reason for placing the power supply so close to the display header. When the layout was complete, the design rule check (DRC) was run to make sure that the PCB layout passed the design rules set by the PCB manufacturer. A design rule file was imported into Altium Designer from JLCPCB, the PCB manufacturer that was chosen to manufacture the board. Once the PCB layout passed all DRC checks, the Gerber files were exported from Altium Designer and an order for ten PCBs was placed with JLCPCB. The turnaround time from order placement to delivery was eight days. After placing an order with Digi-Key for the components and soldering two complete boards, the electronic hardware portion of the project was complete. One of the completed boards is shown in Fig. 8 below. An overview of the board and its components is provided in Fig. 9 and Fig. 10.

Bill of Materials

The bill of materials (BOM) for one fully assembled PCB is shown in Table 1. The total cost for each PCB may not be exactly accurate, as the Digi-Key order that was placed was for three complete boards. Therefore, price breaks occurred in the bulk order. Furthermore, Table 1 does not include the cost for shipping. For a total of ten PCBs from JLCPCB, the cost was $32.20. Therefore, the unit cost is $3.22. This could also prove to be an inaccuracy to the cost of only one board. A deeper financial analysis would have to be performed to better estimate the cost of each board. Nonetheless, a decent estimate still exists.

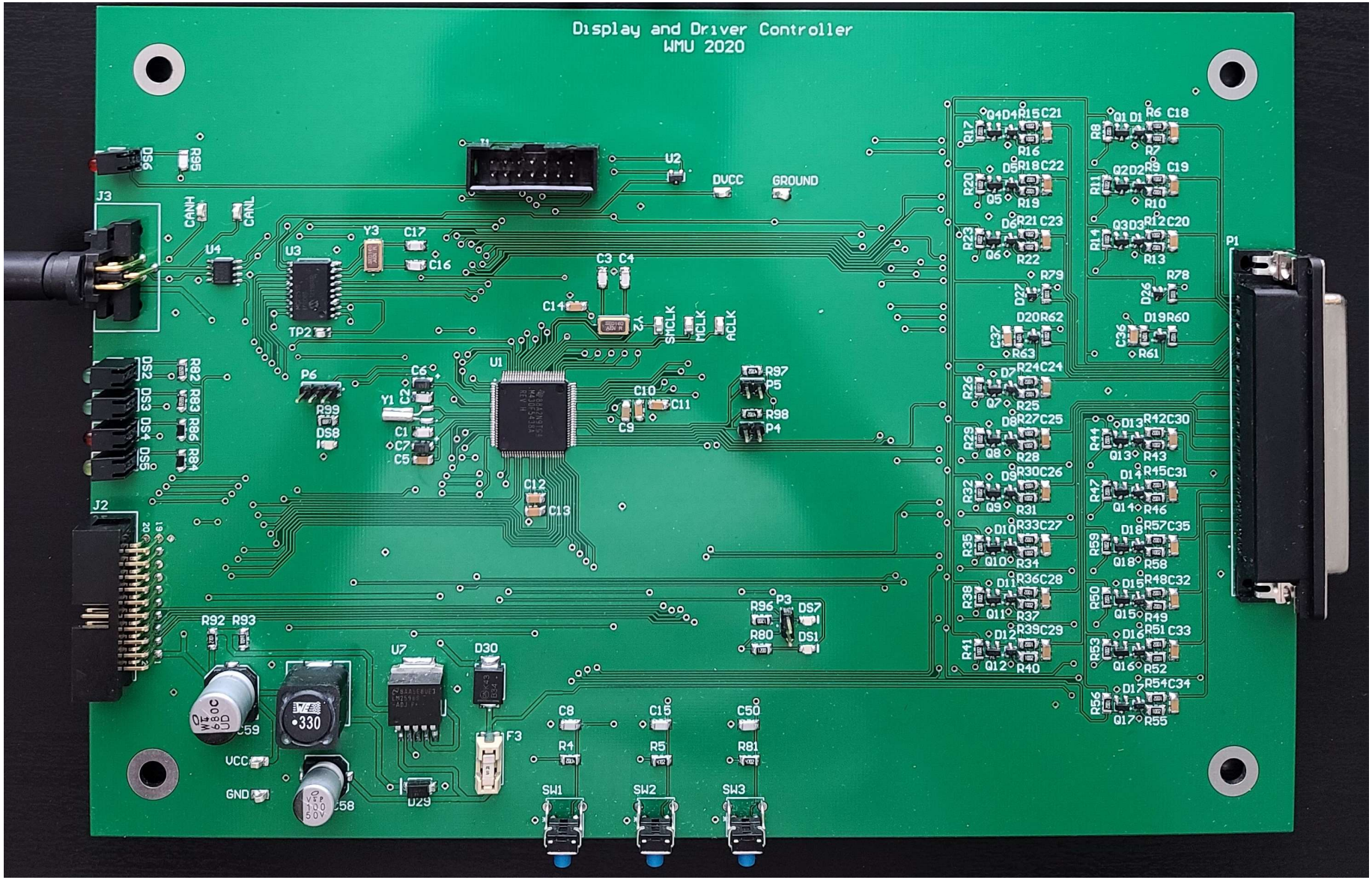


Figure 8: Completed PCB

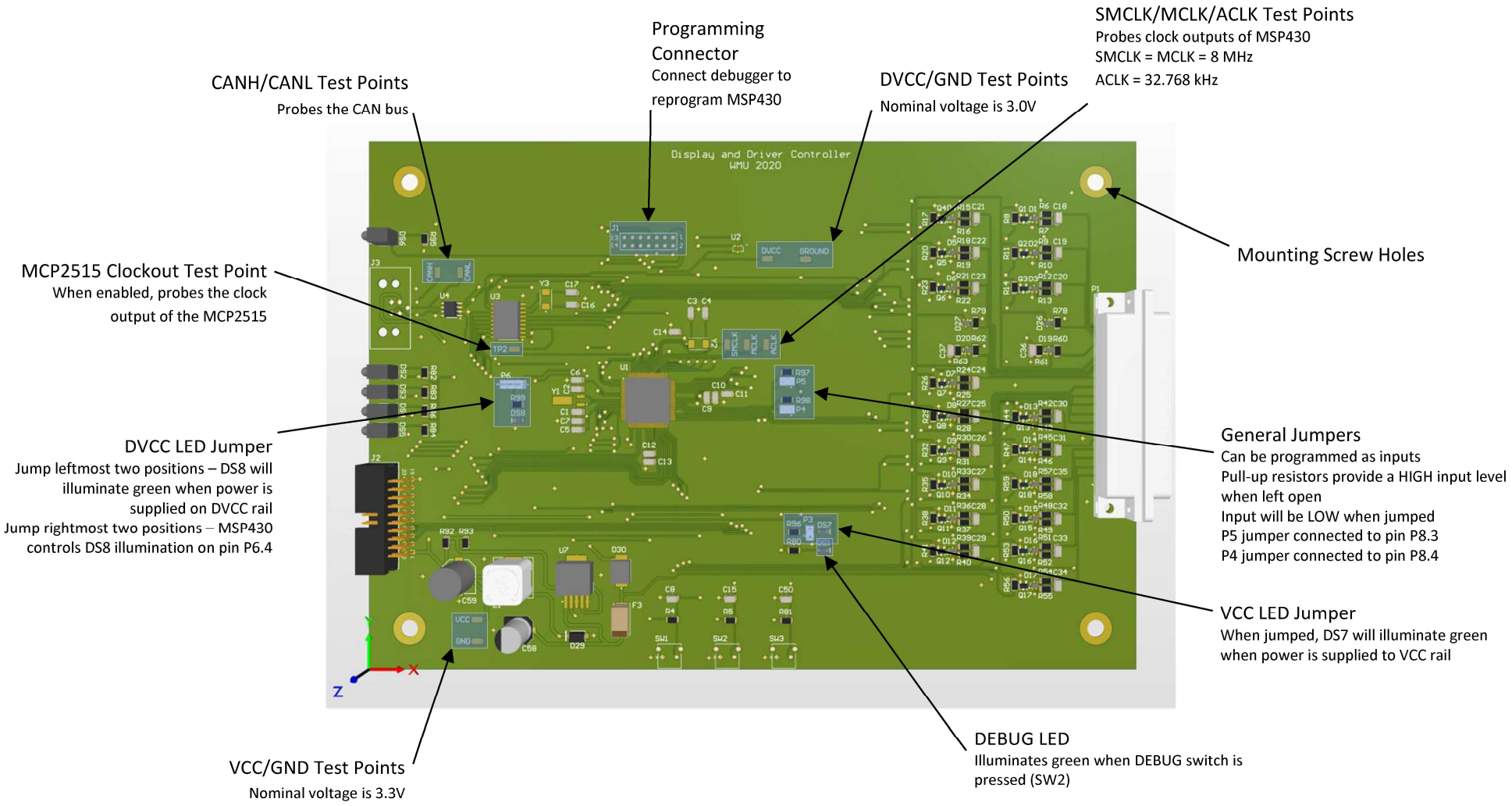
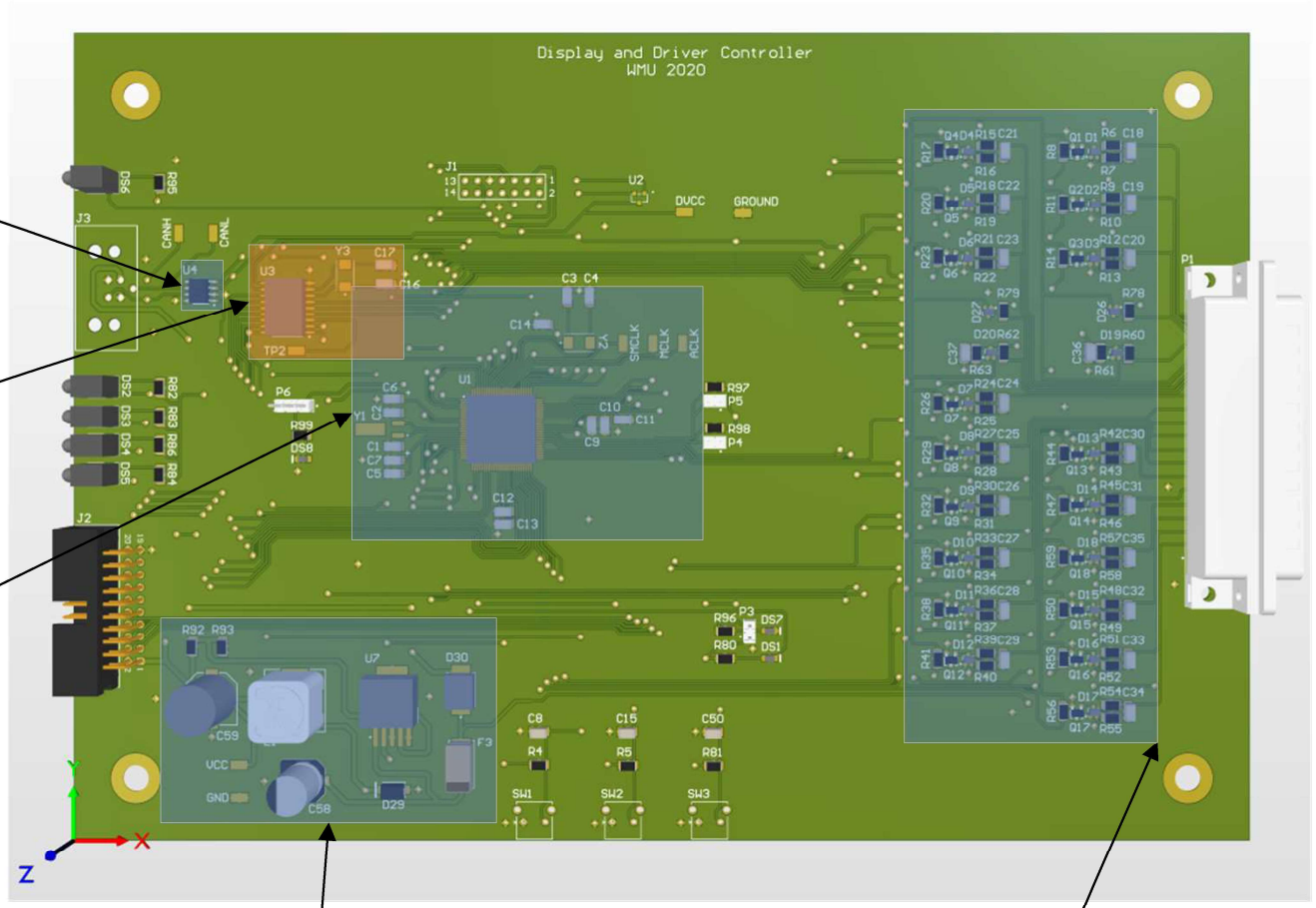


Figure 9: PCB Component Overview Page 1

SN65HVD235D CAN Transceiver
Converts differential CANH/CANL signals to TX/RX signals for MCP2515
Connected directly to the CAN bus

MCP2515 CAN Controller
Processes all CAN logic between the CAN bus and MSP430
Communicates with CAN bus through CAN transceiver (TX/RX signals)
Communicates with MSP430 through an SPI interface
Needs an external high-frequency crystal for operation
Can be programmed to output a divided clock to TP2

MSP430F5438A
Main processor of driver controller PCB
Needs bypass capacitors for operation
External crystals are used instead of internal clock for a more accurate clocking system



Power Supply
Buck converter that steps down 13.8V from CAN bus to 3.3V/3.0V for display and MSP430, respectively. 3.3V output placed close to the display header to prevent voltage drop in trace because of high current draw.

Digital/Analog Circuitry
Protects MSP430 from high voltage
Drops 12V used by switches to 3.3V for MSP430

Figure 10: PCB Component Overview Page 2

Index	Designator	Quantity	Digi-Key Part Number	Description	Unit Price	Extended Price
1	C1, C2	2	399-1193-1-ND	CAP CER 12PF 50V C0G/NP0 1206	0.27	0.54
2	C3, C4, C16, C17	4	399-1194-1-ND	CAP CER 15PF 50V C0G/NP0 1206	0.184	0.74
3	C5, C10, C11, C12, C13, C14	6	399-1250-1-ND	CAP CER 0.1UF 50V X7R 1206	0.219	1.31
4	C6, C7	2	511-1463-1-ND	CAP TANT 10UF 20% 10V 1206	0.53	1.06
5	C8, C15, C50	3	399-8171-1-ND	CAP CER 2200PF 50V C0G/NP0 1206	0.336	1.01
6	C9	1	399-5771-1-ND	CAP CER 0.47UF 50V X8L 1206	0.9	0.90
7	C18, C19, C20, C21, C22, C23, C24, C25, C26, C27	20	399-1250-1-ND	CAP CER 0.1UF 50V X7R 1206	0.1239	2.48
8	C58	1	493-2226-1-ND	CAP ALUM 100UF 20% 50V SMD	0.51	0.51
9	C59	1	493-2267-1-ND	CAP ALUM 680UF 20% 16V SMD	0.89	0.89
10	All Test Points	7	36-5015CT-ND	PC TEST POINT MINIATURE	0.382	2.67
11	D1, D2, D3, D4, D5, D6, D7, D8, D9, D10, D11, D1	22	BAV99WT1GOSCT-ND	DIODE ARRAY GP 100V 215MA SC70-3	0.0556	1.22
12	D29	1	B340A-FDICT-ND	DIODE SCHOTTKY 40V 3A SMA	0.41	0.41
13	D30	1	MBRS340T3GOSCT-ND	DIODE SCHOTTKY 40V 4A SMC	0.47	0.47
14	DS1, DS7	2	516-1434-1-ND	LED GREEN DIFFUSED CHIP SMD	0.51	1.02
15	DS2, DS3	2	350-1817-ND	LED 3MM LOW CURR GREEN	1.36	2.72
16	DS5	1	350-1818-ND	LED 3MM LOW CURR YELLOW	1.36	1.36
17	DS4, DS6	2	350-1816-ND	LED 3MM LOW CURR RED	1.36	2.72
18	J2	1	732-2100-ND	CONN HEADER R/A 20POS 2.54MM	1.04	1.04
19	L1	1	732-1244-1-ND	FIXED IND 33UH 4.2A 45 MOHM SMD	2.41	2.41
20	P1	1	182-37FE-ND	CONN D-SUB RCPT 37POS R/A SOLDER	3.63	3.63
21	Q1, Q2, Q3, Q4, Q5, Q6, Q7, Q8, Q9, Q10, Q11, Q1	18	2N7002NCT-ND	MOSFET N-CH 60V 115MA SOT-23	0.1078	1.94
22	R4, R5, R81, R97, R98	5	311-47.0KFRCT-ND	RES SMD 47K OHM 1% 1/4W 1206	0.071	0.36
23	R6, R7, R8, R9, R10, R11, R12, R13, R14, R15, R1	56	311-10.0KFRCT-ND	RES SMD 10K OHM 1% 1/4W 1206	0.0291	1.63
24	R61, R63	2	311-1.00MFRCT-ND	RES SMD 1M OHM 1% 1/4W 1206	0.1	0.20
25	R78, R79	2	YAG5901CT-ND	RES SMD 1206 1/2W 1% 330 OHM	0.56	1.12
27	R80, R96	2	YAG3810CT-ND	RES SMD 120 OHM 1% 1/4W 1206	0.1	0.20
28	R82, R83	2	311-665FRCT-ND	RES SMD 665 OHM 1% 1/4W 1206	0.1	0.20
29	R84	1	541-750FCT-ND	RES SMD 750 OHM 1% 1/4W 1206	0.1	0.10
30	R86, R95	2	541-820FCT-ND	RES SMD 820 OHM 1% 1/4W 1206	0.1	0.20
31	R92	1	311-2.70KFRCT-ND	RES SMD 2.7K OHM 1% 1/4W 1206	0.1	0.10
32	R93	1	RMCF1206FT1K60CT-ND	RES 1.6K OHM 1% 1/4W 1206	0.1	0.10
33	SW1, SW2, SW3	3	450-1657-ND	SWITCH TACTILE SPST-NO 0.05A 24V	0.17	0.51
34	U1	1	296-37147-1-ND	IC MCU 16BIT 256KB FLASH 100LQFP	6.81	6.81
35	U2	1	BAT54C-FDICT-ND	DIODE ARRAY SCHOTTKY 30V SOT23-3	0.19	0.19
36	U3	1	MCP2515-I/SO-ND	IC CAN CONTROLLER W/SPI 18SOIC	1.82	1.82
37	U4	1	296-15229-5-ND	IC TRANSCEIVER HALF 1/1 8SOIC	3.13	3.13
38	U7	1	296-35399-1-ND	IC REG BUCK ADJ 3A TO263-5	4.77	4.77
39	Y1	1	XC1911CT-ND	CRYSTAL 32.7680KHZ 12.5PF SMD	0.72	0.72
40	R99	1	YAG3909CT-ND	RES SMD 88.7 OHM 1% 1/4W 1206	0.1	0.10
41	Y2, Y3	2	XC983CT-ND	CRYSTAL 16.0000MHZ SERIES SMD	1.12	2.24
42	F3	1	F1222CT-ND	FUSE BRD MNT 1A 125VAC/VDC 2SMD	2.55	2.55
43	J1	1	A101791-ND	CONN HEADER VERT 14POS 2.54MM	1.72	1.72
44	J3	1	277-2689-ND	CONN PLUG MALE 5POS GOLD SOLDER	23.8	23.80
45		1		Fabricated PCB	3.22	3.22
					Total	$86.84

Table 1: Bill of Material for one PCB

Mechanical Design

Overview

Both the display and the PCB required a housing to make mounting them in the solar car easier, and for protection. Using PTC Creo Parametric, a box was designed to hold the PCB with cutouts for the external connector, LEDs, and switches. The box was designed to be held together with four screws that are inserted from the top of the enclosure. An overview of the encosure is shown in Fig. 11 below.

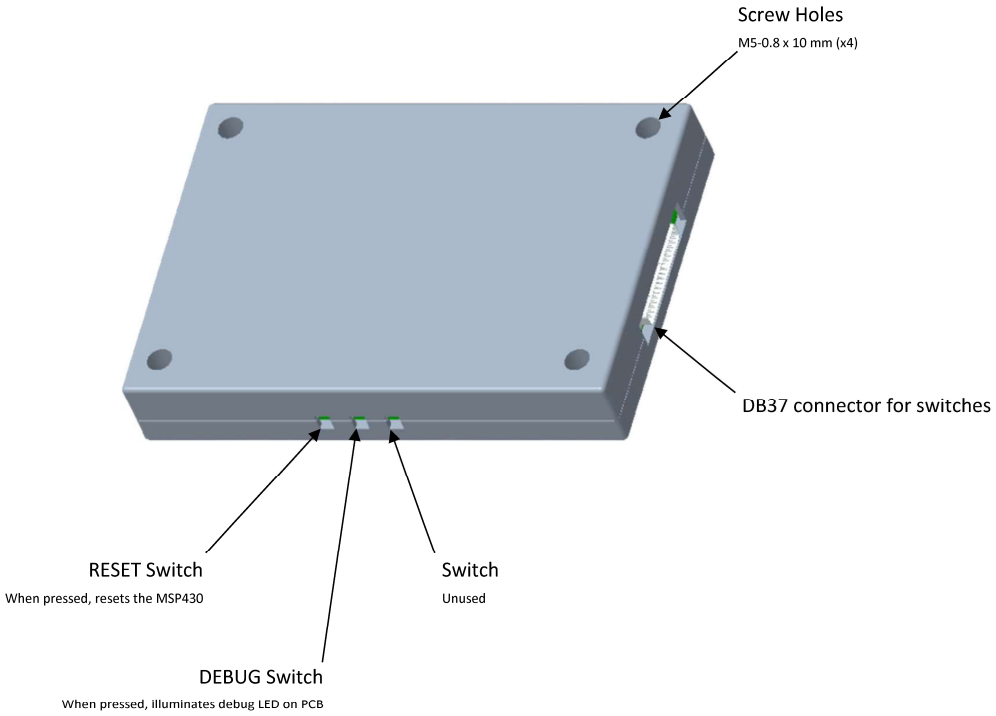


Figure 11: PCB Enclosure Overview (Top View)

An overview of the connectors on the left side of the box is shown in Fig. 12 below.

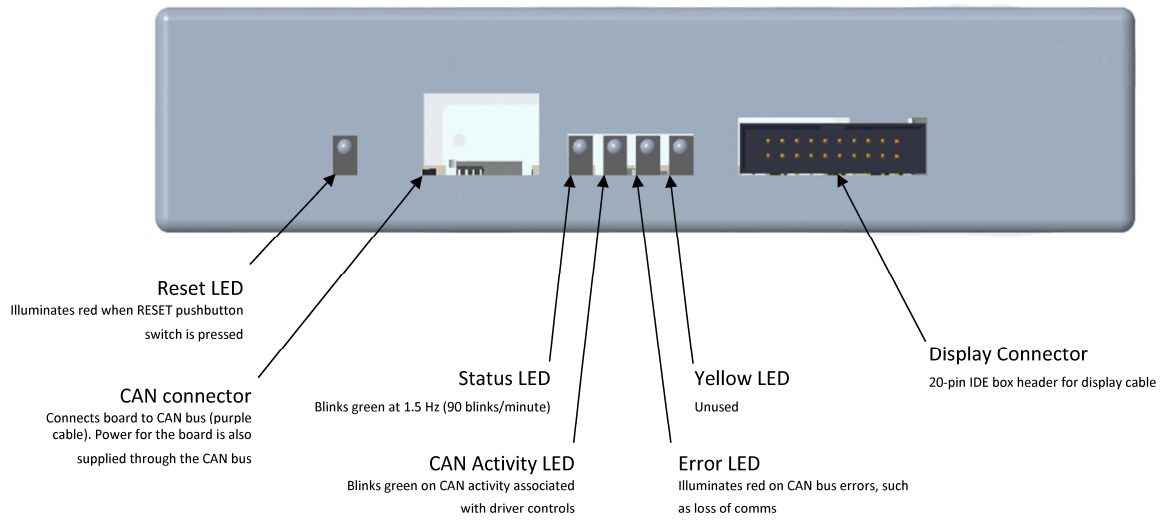


Figure 12: PCB Enclosure Overview (Left View)

The holder for the display was designed to completely encapsulate the display, with a cutout on the top of the display for the screen and a cutout of the rear of the enclosure for the connector. Four screws are meant to hold the two separate pieces together, which go all the way through the two pieces and stick out on the rear of the enclosure. Lock washers and knurled nuts are meant to be placed on the exposed screws to hold the enclosure together. An overview for the display's enclosure is shown in Fig. 13 below.

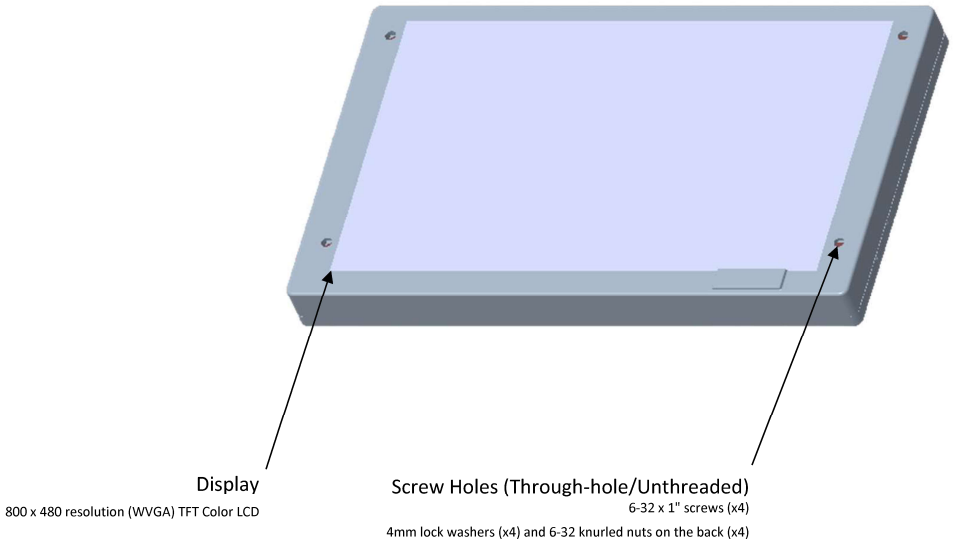


Figure 13: Display Enclosure Overview (Top View)

An overview of the connector on the rear of the enclosure is shown in Fig. 14 below.

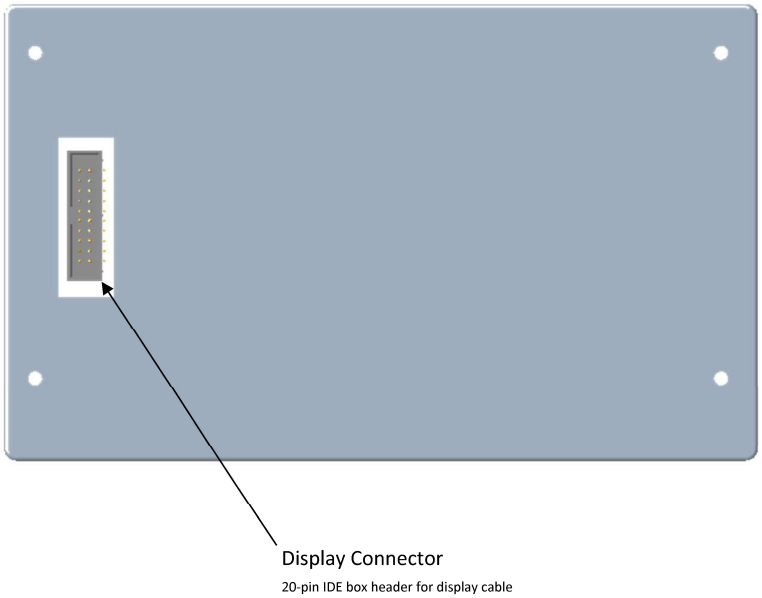


Figure 14: Display Enclosure Overview (Rear View)

Both enclosures were 3D printed using exported STEP files from PTC Creo Parametric. The display enclosure fit the display on the first 3D print, however, the enclosure for the PCB was modified and reprinted after the first version proved to be too small for the PCB. This is because

the inside of the box was designed with the exact dimensions of the PCB. Therefore, a tenth of an inch was added as a tolerance in the next version of the 3D print.

Dimensions

A dimensioned version of the PCB enclosure is shown below in Fig. 15 and Fig. 16.

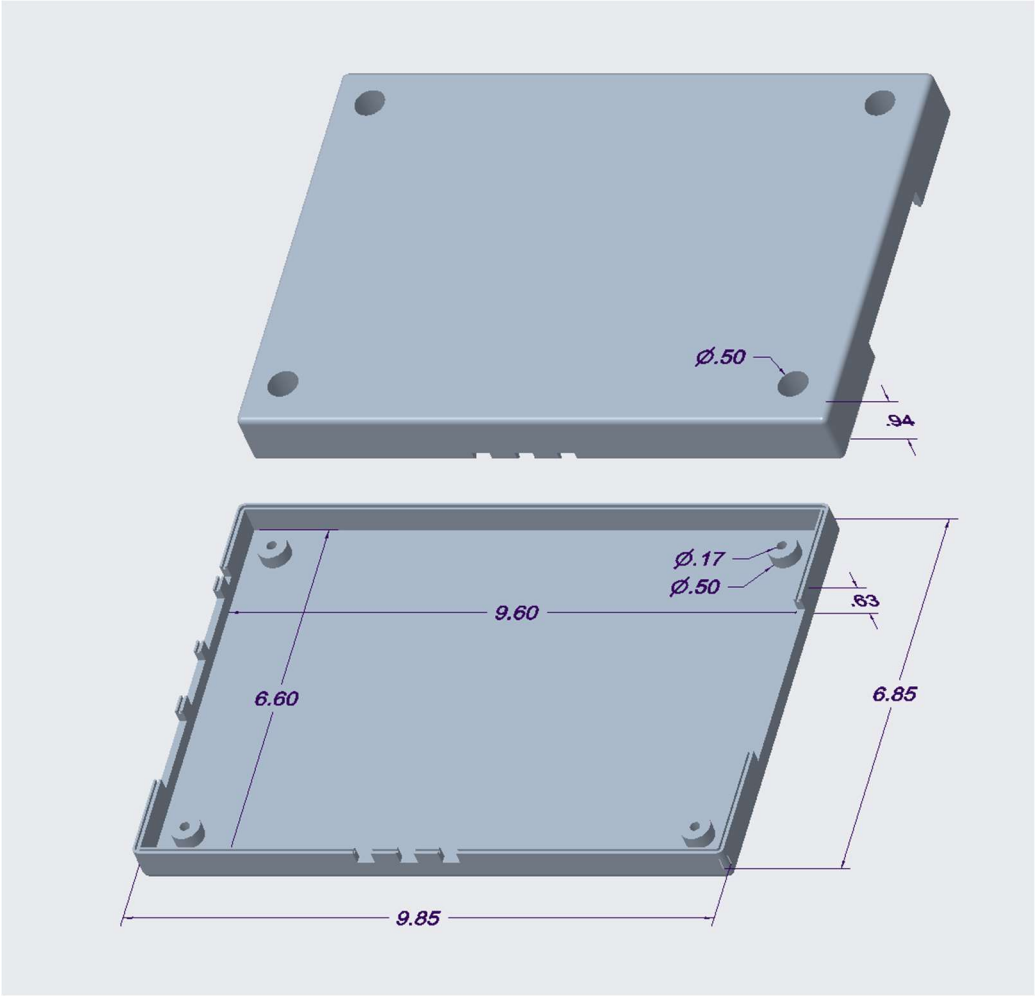


Figure 15: PCB Enclosure Dimensions

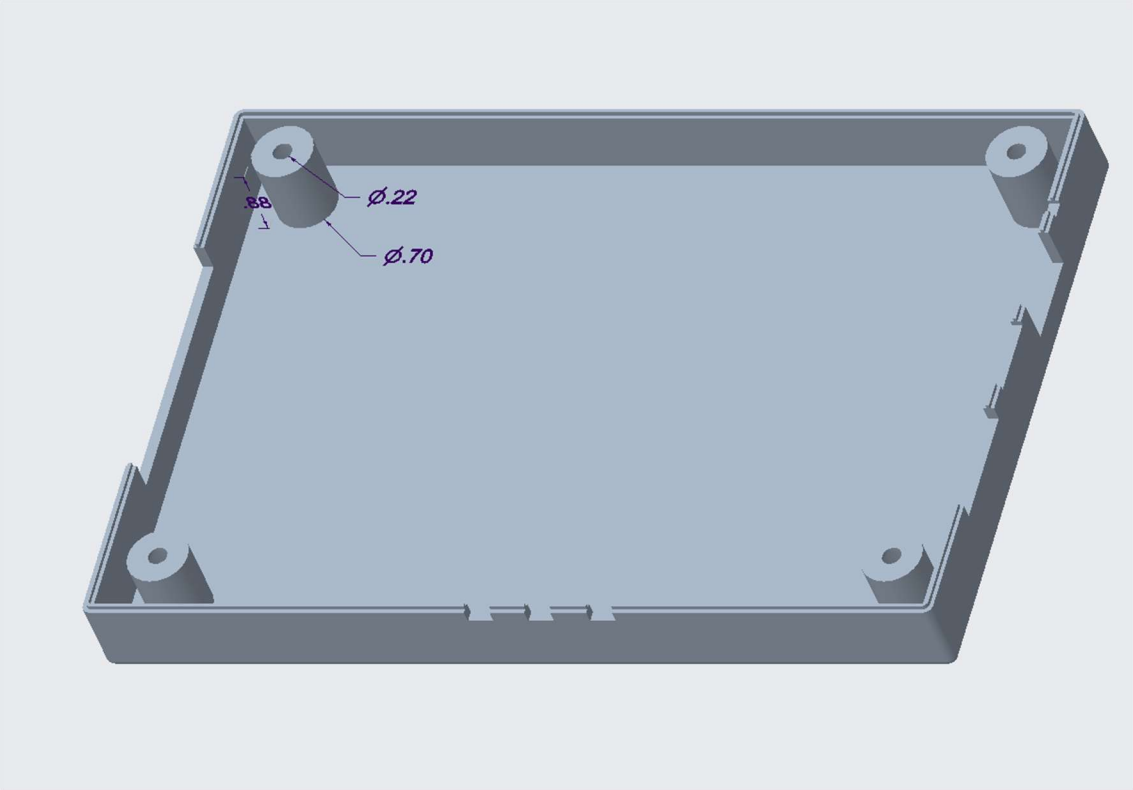


Figure 16: PCB Enclosure Dimensions (Top Piece)

A dimensioned version of the display holder is shown below in Fig. 17.

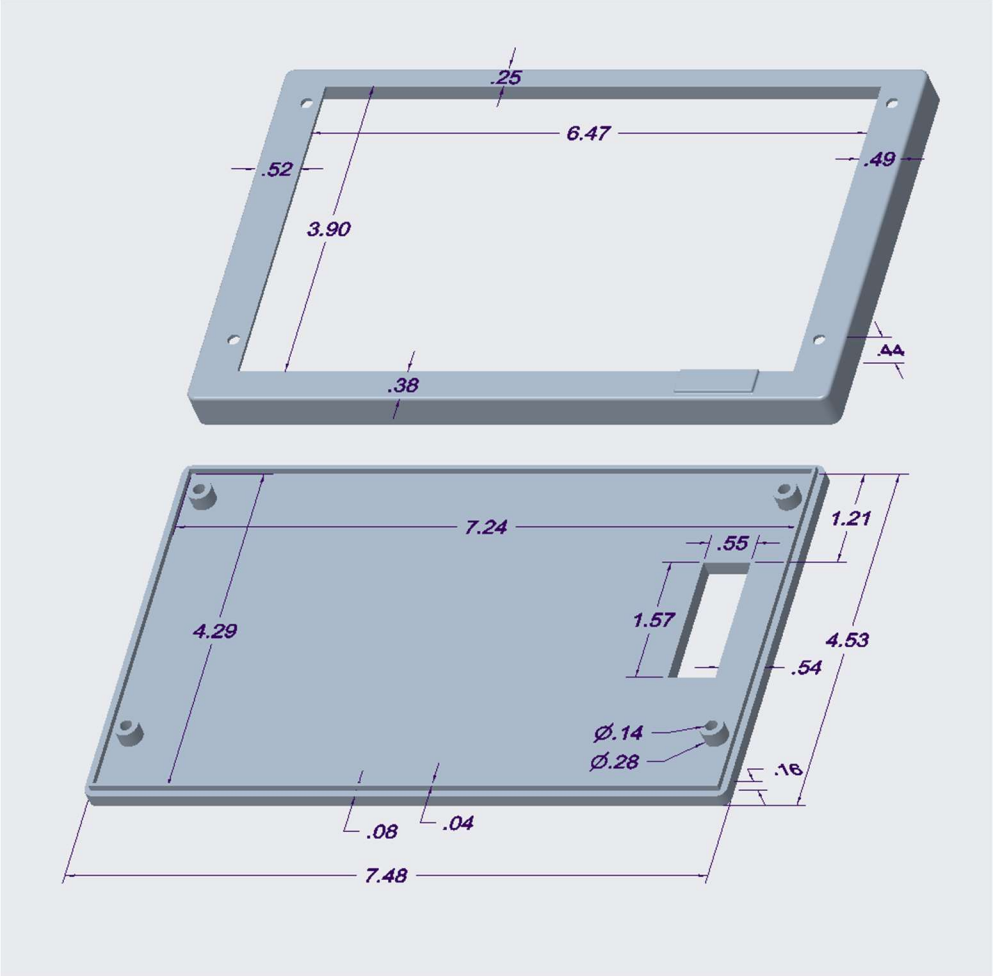


Figure 17: Display Enclosure Dimensions

3D Printed Enclosures

Pictures of the 3D printed enclosures are shown in Appendix F.

Software Design

Introduction

The software for the display and the driver controller is written in the C programming language. When the display was prototyped, its own set of code was created separately from the driver controller code. Then, the code for both segments of the project were combined in Code Composer Studio, an integrated development environment (IDE) specifically created for Texas Instruments' line of processors. Because there is already an existing version of the driver controller, the code was taken and ported to the MSP430F5438A. The existing driver controller used a different variant of the MSP430 and had less pins. Therefore, the driver controller code had to be modified slightly to fit the new variant of the MSP430. Because these changes are very minor and the driver controller code already existed, most of the software for the driver controller will not be described in this document. The only software that will be described consists of the CAN messages that are received and sent by the new DDC. Therefore, the bulk of the content in this section pertains to the software written for the display and the information that needs to be understood so that a member of the solar car team can add new measurements to the screen or alter the layout of the screen.

As mentioned in the hardware design section of this report, the display consists of the panel and the FT812 display controller. The FT812 display controller is the device that the MSP430 communicates with directly to send display commands. The MSP430 never communicates directly with the panel. The communication interface between the FT812 and the MSP430 consists of a 4-line SPI interface with an SPI clock frequency of 8 MHz. The MSP430 consists of three main clocks, and the SPI clock frequency can be programmed to be a prescaler

of any two of the three main clocks. Table 2 shows the various clocks of the MSP430 and their corresponding frequencies.

Table 2: MSP430F5438A Clock Frequencies

Clock	Frequency
MCLK	8 MHz
SMCLK	8 MHz
ACLK	32.768 kHz

The MCLK is the master clock of the microprocessor and is used directly by the CPU and the system. The SMCLK and the ACLK are the sub-main clock and the auxiliary clock, respectively, and are selectable to be used by individual peripheral modules, such as the SPI module. The DDC PCB contains a 16 MHz high frequency crystal and a 32.768 kHz watch crystal. The 16 MHz external crystal is prescaled with a factor of 2 and used as the source for both the MCLK and the SMCLK. The external watch crystal is used as the source for the ACLK directly, without any prescaler. The SPI clock for the FT812 SPI module uses the SMCLK directly, which is why the SPI clock frequency is 8 MHz.

FT812 Display Controller

The FT812 is a little-endian memory mapped device, which means that the contents that are meant to be displayed on the panel are simply written to the FT812's memory, with the least significant bit always being stored at the lowest address. Once all graphics commands have been written, a command is then sent to the FT812 to refresh the display with the contents of its memory. All data and commands are sent over the SPI interface. Therefore, there exists multiple layers of abstraction in the software. There are functions that represent the actual display command, such as the drawing of a gauge. This command calls the memory-write functions, which write the command into the FT812's memory. These memory-write functions call the spi-

transmit functions, which are responsible for transmitting the data over the SPI interface. This process is repeated for each display element on the screen.

Frame Layout

The current layout of the screen is displayed in Fig. 18 below.

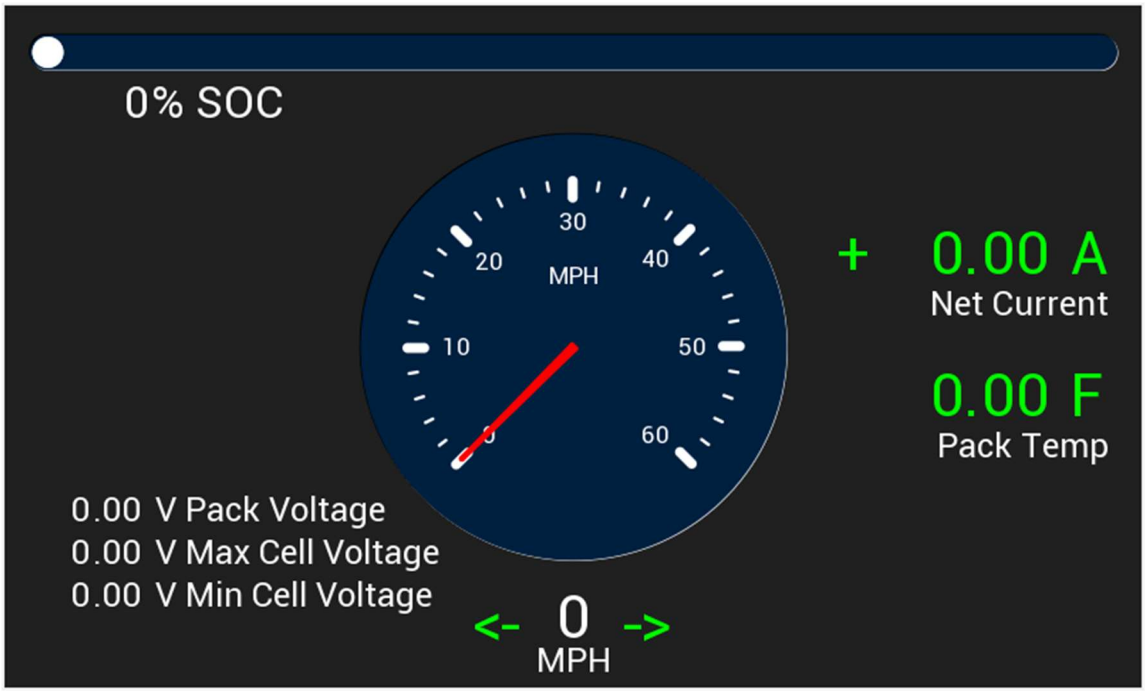


Figure 18: Current Screen Layout

As Fig. 18 shows, the current active measurements consist of the following:

1. Speedometer
    a. Displays the current speed of the solar car in miles-per-hour
    b. Uses the GAUGE, TEXT, COLOR_RGB, and NUMBER commands
2. Maximum cell voltage
    a. Displays the voltage value for the cell that has the maximum voltage
    b. Uses the TEXT and NUMBER commands

3. Minimum cell voltage
    a. Displays the voltage value for the cell that has the minimum voltage
    b. Uses the TEXT and NUMBER commands
4. Net current
    a. Displays the net current of the battery. A positive value will be displayed in green with a '+' sign and a negative value will be displayed in red with a '-' sign
    b. Uses the COLOR_RGB, TEXT, and NUMBER commands
5. Pack temperature
    a. Displays the temperature value for the cell that has the maximum temperature
    b. Uses the COLOR_RGB, TEXT, and NUMBER commands
6. Turn Signals and Hazard
    a. Displays the left or right turn signal, or both for a hazard
    b. Uses the VERTEX command

The current inactive measurements consist of the following:

1. Pack Voltage
    a. Displays the current overall voltage of the battery
    b. Uses the TEXT and NUMBER commands
2. State of charge
    a. Displays the current charge of the battery as a percent of the total battery capacity
    b. Uses the PROGRESS, TEXT, and NUMBER commands

The pack voltage and state of charge are currently inactive measurements because they share the same CAN bus message. Another CAN bus message must be added to the Battery Protection System (BPS) to support both measurements.

EVE Screen Editor

The layout for the screen was created in the EVE Screen Editor software, which is a software package provided by the manufacturer of the display to support prototyping of the display. This software package as well as other utilities can be obtained from the following link:

https://www.ftdichip.com/Support/Utilities.htm

Fig. 19 shows a screenshot of the EVE Screen Editor Software.

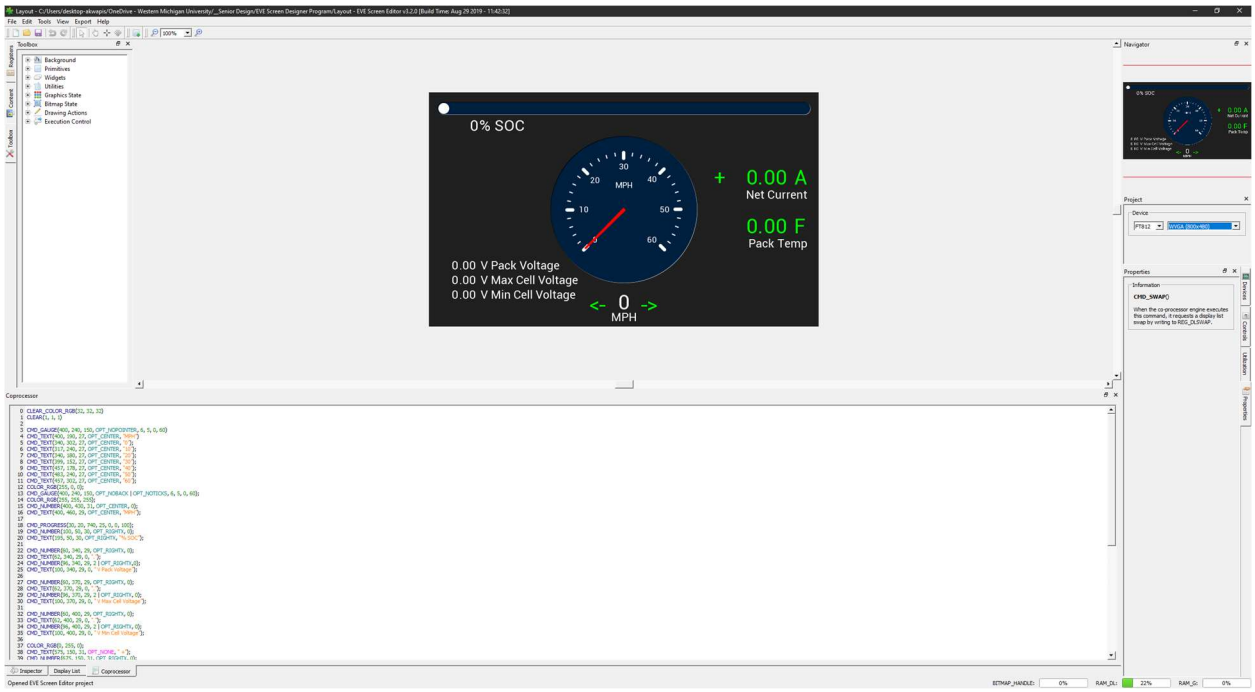


Figure 19: EVE Screen Editor Software

As shown in Fig. 19, the environment consists of the frame to be displayed on the screen as well as a list of commands that correspond to elements on the screen. The command list is not written in the C language, it is only a list of commands with parameters. These commands must be

slightly modified when transferring them to the C code, which includes appending the text "EVE_" to the beginning of each command and changing the zero value of the command to the desired value. The project in the EVE Screen Editor software that has the current screen's layout will be electronically delivered to the Sunseeker solar car team for their future use.

Adding New Measurements to the Display

The following sequence is the recommended way to add a new measurement from the CAN bus to the display.

1. Configure the CAN filters and masks of the MCP2515 (CAN controller) to accept the CAN message that has the value to displayed on the screen. This will be expanded on later in this report.
2. At the top of the EVE.h header file, declare the variable for the specific measurement to be added to the screen, however, do not initialize this variable.
3. At the top of the main.c file (under 'Display values'), add two variables for the new measurement, one variable to hold an old value of the measurement and one variable to hold the new value of the measurement. The new value will be set when the MSP430 receives the specific CAN message with the desired value. The old value will be set to the new value when the display is updated with the new value. Therefore, the display is only ever updated if the new value is different from the old value.
4. Add a case statement to the switch statement that begins when the MSP430 checks if the CAN_nINT input is low. This occurs around line 785. The code corresponding to this new case statement must set the new value of the variable declared in step 3 above. Make sure that the right value from the CAN bus is sampled, there is a HIGH word and a LOW word.

5. In the section where the MSP430 handles communication events (above the aforementioned switch statement), add an if statement to compare the new value and the old value declared in step 3 above. Update the value declared in step 2 above with the new value declared in step 3 above. Set the old value to the new value for the next time around the loop. Set the 'needUpdate' variable to 'true' so that the display knows it needs to be refreshed.
6. Design the measurement in the EVE Screen Editor program and port the related commands over to the EVE_BuildFrame function in the EVE.c file. Now, whenever the MSP430 receives the specified CAN message, the measurement on the screen will be updated with the value from the CAN bus if its value changed from a previous reception.

Display Commands

Each measurement on the display can have a number of commands that define the way the measurement will look or function on the display. Each individual command is a series of bytes that are written to the FT812's memory. The FT81X Programmers Guide lists all of the available commands in chapter 5 [1]. However, only the commands that were used in this project will be referenced. The main commands that were used in this project are presented in Appendix A, and their corresponding C function is located in the EVE.c file under the 'EVE commands' section. As an example, the GAUGE command will be examined, both from the FT81X Programmers Guide, and the documentation that appears in Appendix A. The recreated documentation from Appendix A is shown in Fig. 20 below. The original pages from the FT81X Programmers Guide appear in Fig. 21 and Fig. 22.

EVE_CMD_GAUGE
Found on page 187 of FT812 Programmer Guide

Encoding

Offset	Parameter	Length
+0	CMD_GAUGE (0xFFFFFF13)	4 bytes
+4	X	2 bytes
+6	Y	2 bytes
+8	R	2 bytes
+10	Options	2 bytes
+12	Major	2 bytes
+14	Minor	2 bytes
+16	Value	2 bytes
+18	Range	2 bytes

Parameters

X - X-coordinate of gauge center, in pixels
Y - Y-coordinate of gauge center, in pixels
R - Radius of the gauge, in pixels
Options - By default the gauge dial is drawn with a 3D effect and the value of options is zero.
OPT_FLAT removes the 3D effect. With option OPT_NOBACK, the background is not drawn. With option OPT_NOTICKS, the tick marks are not drawn. With option OPT_NOPOINTER, the pointer is not drawn.
Major - Number of major subdivisions on the dial, 1-10
Minor - Number of minor subdivisions on the dial, 1-10
Value - Gauge indicated value, between 0 and range, inclusive
Range - Maximum value

Description

Draws a gauge widget. The total length of the command is 20 bytes.

Figure 20: Recreated documentation of the GAUGE command

5.33 CMD_GAUGE - draw a gauge

C prototype

```
void cmd_gauge( int16_t x,
                int16_t y,
                int16_t r,
                uint16_t options,
                uint16_t major,
                uint16_t minor,
                uint16_t val,
                uint16_t range );
```

Parameters

x

X-coordinate of gauge center, in pixels

y

Y-coordinate of gauge center, in pixels

r

Radius of the gauge, in pixels

options

By default the gauge dial is drawn with a 3D effect and the value of options is zero. OPT_FLAT removes the 3D effect. With option OPT_NOBACK, the background is not drawn. With option OPT_NOTICKS, the tick marks are not drawn. With option OPT_NOPOINTER, the pointer is not drawn.

Figure 21: Original datasheet description of the GAUGE command Page 1 [1]

major

Number of major subdivisions on the dial, 1-10

minor

Number of minor subdivisions on the dial, 1-10

val

Gauge indicated value, between 0 and range, inclusive

range

Maximum value

Description

The details of physical dimension are:

- The tick marks are placed on a 270 degree arc, clockwise starting at south-west position
- Minor ticks are lines of width r*(2/256), major r*(6/256)
- Ticks are drawn at a distance of r*(190/256) to r*(200/256)
- The pointer is drawn with lines of width r*(4/256), to a point r*(190/256)from the center
- The other ends of the lines are each positioned 90 degrees perpendicular to the pointer direction, at a distance r*(3/256) from the center

Refer to Co-processor engine widgets physical dimensions for more information.

Command layout

+0	CMD_GAUGE(0xffffff13)
+4	X
+6	Y
+8	R
+10	Options
+12	Major
+14	Minor
+16	Value
+18	Range

Figure 22: Original datasheet description of the GAUGE command Page 2 [1]

As seen from either Fig. 20, Fig. 21, and Fig. 22, the command layout is presented in a series of bytes. The first four bytes of the command is what defines it as the GAUGE command. These four bytes are somewhat of an 'opcode' to the FT812; it knows once it sees these four bytes what parameters the following bytes will correspond to. There are a number of parameters for the gauge, such as location (X and Y parameters, referenced from the top left of the screen), the radius of the gauge (R parameter), and the actual value to be displayed (value parameter). All these different parameters have their own byte offsets in the command. The total length of the command is 20 bytes long, which never changes. The TEXT command, for example, is a variable-length command, as each character to be printed on the screen consumes one byte of memory. The layout of the commands that have already been implemented in the display software do not need to be fully understood. However, it is recommended that time is taken to understand how each command was interpreted from the FT81X programmers guide into a C function. This way, more commands can be added in the future with ease. Each row of the command encoding (or layout) must be written to the FT812's memory separately. This is done by the memory-write functions. The memory-write functions are called multiple times in the GAUGE command function, one function call for each parameter. In fact, this is true for all the commands that have implemented.

Types of SPI Transactions

As mentioned earlier, the memory-write functions call the spi-transmit functions. The SPI transactions must occur following a set of guidelines, so that the FT812 can decipher a memory-write command from a memory-read command, or from a host command. These are the three types of commands sent over SPI, with the memory-write command being used the most often and during the writing of display commands to memory. The memory-read command is only

used during initialization of the display, which will be described later in this document. The host command is also only used during initialization of the display. The layout of these three types of commands in shown in Appendix B. For example, a memory-write function is executed over SPI by sending a 3-byte address with the two most-significant bits set to 2'b10. When the FT812 sees these two bits, it knows that the following SPI transaction will be a memory-write. After the 3-byte address has been sent, the data is transmitted byte-by-byte. Any number of bytes can be sent because the internal address pointer of the FT812 is automatically incremented after each byte. The memory-read functions are executed in a similar manner, but by sending 2'b00 as the two most significant bits and sending a dummy byte in between the address and the data to be received by the MSP430. Furthermore, the underlying SPI function for the second half of a memory-read command is the spi-exchange function, not the spi-transmit function. In the spi-exchange function, the MSP430 sends a byte and waits until the entire byte has been sent before sampling the return byte from the FT812. On the other hand, the spi-transmit function does not expect anything in return. A host command layout is also shown in Appendix B.

Memory Read and Write

The diagrams in Appendix B are the underlying templates for the memory-write, memory-read, and host command functions in the C code. For example, there are multiple memory-write functions in the software for the display. There is a function to write a single byte, two bytes, or four bytes. Likewise, there are multiple memory-read functions. There is a function to read a single byte and to read two bytes. There is only one host command function, which only requires the command to be executed by the FT812 to be sent. These are all fixed length commands, which is why only one function is required. The main point here is that all the memory-write functions are based off the host memory-write template and all memory-read

functions are based off the host memory-read template. The different memory-write and memory-read commands are shown in Appendix C.

The GAUGE command presented in Fig. 20, Fig. 21, and Fig. 22 uses the two-byte and the four-byte memory-write functions, which are shown in Fig. C2 and Fig. C3 in Appendix C, respectively. The first four bytes of the GAUGE command (the 'opcode') will use the four-byte memory-write function and the rest of the parameters will use the two-byte memory-write functions. The GAUGE command was only used as an example; however, it shows that all graphics commands that are meant to be displayed on the screen use a form of the memory-write functions, which are all based off the host memory-write template. All memory-read functions are based off the host memory-read template, and the host command function is based off the host command template, both of which are only used in the initialization of the display and will be described later in this document.

Coprocessor Engine

Up to this point, the MSP430's software has been described in detail as to how graphics commands are written to memory using the memory-write functions and the spi-transmit functions. However, not much detail as to where these commands are written or how the display processes these commands has been mentioned. Graphics commands are written to the FT812 coprocessor's memory. The coprocessor is responsible for reading the graphics commands stored in its memory, processing all the commands, and then refreshing the display with the new frame. The coprocessor's memory space is 4 KB (4096 bytes) in size and its starting address in the FT812's memory map is labeled RAM_CMD in the software, which is 0x308000. Graphics commands are written into this memory space starting at the next available memory location.

The first time the MSP430 writes to the coprocessor, it will begin at memory location 0x308000. Conceptually, the coprocessor's memory space is depicted in Fig. 23.

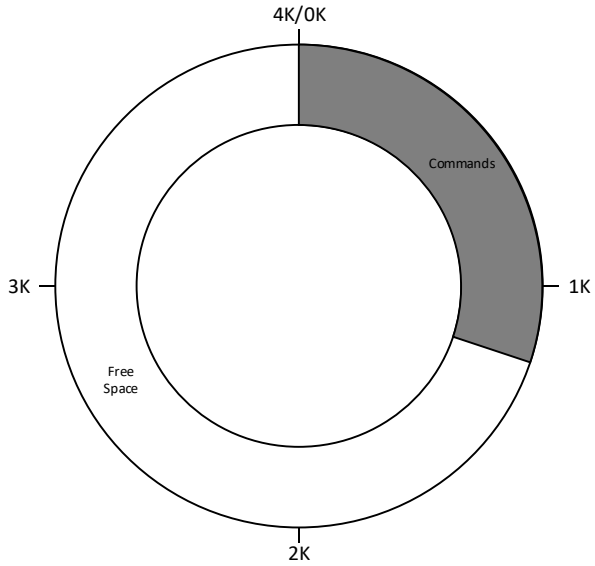


Figure 23: Coprocessor memory space

The coprocessor's memory space is depicted as a circular buffer, as seen in Fig. 23. The importance of this will be mentioned later in this section of the report. Nonetheless, the gray portion of the diagram represents graphics commands written to memory by the MSP430 and the white portion of the diagram represents free space. The MSP430 must keep track of the current offset in the buffer so it knows where to write each graphics command. This variable in the display's software is called 'cmdOffset'. The coprocessor keeps track of two pointers that are used to update the display with the commands stored in memory. REG_CMD_WRITE is a register in the memory of the FT812 (outside of the coprocessor's memory space) that contains the offset of the next available memory location in the buffer. Likewise, REG_CMD_READ is very similar, except it contains the offset of the next graphics command to be processed by the coprocessor. Before any commands are written to the coprocessor, the buffer is empty and both

pointers point at 0x308000 (REG_CMD_WRITE = REG_CMD_READ = 0x308000). A depiction of this is in Fig. 24 below.

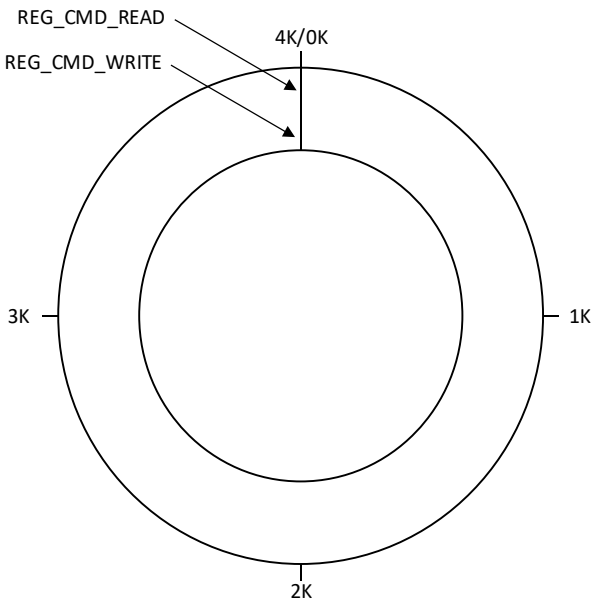


Figure 24: Empty Coprocessor Buffer

Because both pointers are pointing to the same location in the coprocessor's memory, it appears as if the buffer is empty and therefore, the coprocessor has no commands to execute. All commands written into the buffer must be four-byte aligned (a multiple of four). An error occurs in the coprocessor if a complete graphics command is not 4-byte aligned. This is not the case for individual memory-write commands, but for the graphics command as a whole. For example, the PROGRESS command is padded with two extra bytes, which simply prevents an error in the coprocessor and has no meaning graphically. Immediately after commands are written into the buffer, Fig. 25 depicts what the buffer would look like.

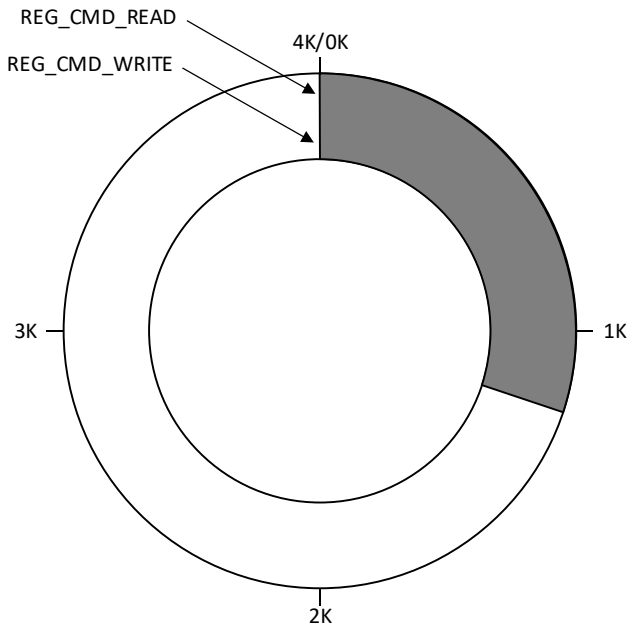


Figure 25: Coprocessor buffer immediately after commands have been written

In order for the coprocessor to execute the new commands stored in its memory, the pointer REG_CMD_WRITE must be updated with the new value of cmdOffset. Because all the commands have been written, cmdOffset contains the offset of the very next location after the last graphics command in memory. Therefore, updating REG_CMD_WRITE with cmdOffset will select all the graphics commands that were just written so they can be processed by the coprocessor. This process is done by calling EVE_UPDATE in the MSP430's code. EVE_UPDATE will write the value of cmdOffset to the register REG_CMD_WRITE. The moment REG_CMD_WRITE is updated, the two pointers are no longer equal, which signifies to the coprocessor that there are commands to be executed. The coprocessor processes each command in memory and increments REG_CMD_READ until the two pointers become equal again. Once the two pointers become equal again, the new frame will be visible on the display. To update the values in the frame, the corresponding values should be changed in the C code and

then the BuildFrame function should be called. This function rewrites the entire frame to the coprocessor's memory in the next available location.

Because the same frame is rewritten to the display over and over, the current frame it will fit in the 4 KB memory space of the coprocessor about five times. When the sixth frame must be written, it will overwrite the first frame at the beginning of the circular buffer. The cmdOffset variable in the MSP430's code accounts for the size of the coprocessor's memory space and will start at zero again once it is incremented past 4095, which is the index of the last location in the coprocessor. On the display's side, the coprocessor is intelligent enough to wrap the REG_CMD_READ pointer around the buffer back to the beginning as it processes the commands. Therefore, the same frame can be rewritten to the coprocessor indefinitely as long as a single frame is less than 4 KB in size.

Display Initialization

In order for the display to begin processing commands, it must be initialized. This involves sending host commands and reading from the FT812's memory to determine if it is ready to receive graphics commands. The MSP430's code has a function that turns on the display and initializes it so that it can receive graphics commands. The flowchart presented in Fig. 26 shows this initialization sequence. Each block in the flowchart has a drilled down diagram that can be found in Appendix D, except for the 'Initialize FT812 Registers' and the 'Build Frame' steps. These flowcharts will be delivered electronically to the Sunseeker solar car team. The purpose of each step will be described briefly.

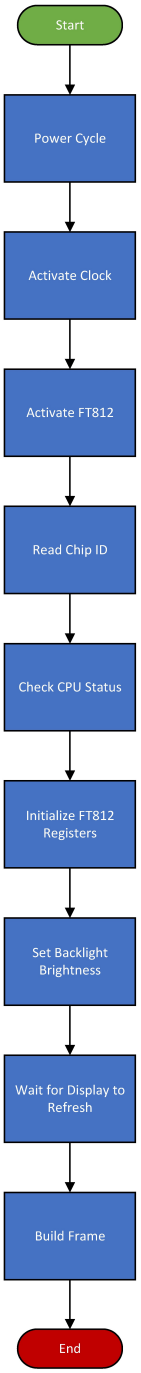


Figure 26: Recommended FT812 Initialization Sequence

FT812 Initialization Sequence

1. Power Cycle (Fig. D2)

   The power cycle sets the EVE_PDN pin (output of MSP430, input to FT812) low for 6 ms, then the same pin is set high. This powers down the FT812 and powers it back up, resetting the chip.

2. Activate Clock (Fig. D3)

   A host command is sent to the FT812 to select the internal clock as the main clock.

3. Activate FT812 (Fig. D4)

   A host command is sent to the FT812 to activate it and turn it on.

4. Read Chip ID (Fig. D5)

   The MSP430 continuously reads the REG_ID register until its value is 0x7C. When this occurs, the FT812 has completed its startup sequence. This function will timeout after 2000 consecutive reads.

5. Check CPU Status (Fig. D6)

   The MSP430 continuously reads the REG_CPURESET register until both the coprocessor engine and the touch engine are ready. This function will timeout after 2000 consecutive reads.

6. Initialize FT812 Registers

   The MSP430 writes resolution-specific values to several different registers to configure the FT812 to process commands at a specific resolution.

7. Set Backlight Brightness (Fig. D7)

   The MSP430 turns on the panel's backlight to 15% brightness by writing to the REG_PWM_DUTY register.

8. Wait for Display to Refresh (Fig. D8)

   Waits until REG_CMD_READ is equal to REG_CMD_WRTIE. When this occurs, the coprocessor engine is ready to receive graphics commands.

9. Build frame

   The first frame is written to the coprocessor. All measurements have a value of zero. After this function call, the frame becomes visible on the display.

Using this recommended process, the display is initialized, and the first frame is written to the display. Every subsequent display refresh uses the same function in the MSP430's code to rewrite the same frame to memory and refresh the display. This not only saves code but makes it very easy to change the variable for a measurement in the code and have the value updated on the display.

Transmitted CAN Messages

The driver controller sends four different messages to the CAN bus, which are expected by the other devices on the bus. The layout of these four commands can be found in Appendix E. The DC_CAN_BASE message, shown in Fig. E1, contains a serial number in the high word and the string "TRIb" in the low word. In all the messages that the driver controller transmits, the high word is always sent to the CAN bus first. This means that when the CAN bus is probed, the most significant four bytes of the data field will be the high word and the least significant four bytes will be the low word. Within each word, the least significant byte is sent first, and within each byte, the most significant bit is sent first. The DC_CAN_BASE message has an ID of 0x500 and is sent over the CAN bus once every second. Fig. E1 shows the layout of this message bit-by-bit.

The DC_DRIVE message is shown in Fig. E2. Its ID on the CAN bus is 0x501 and the message is sent over the CAN bus every 100 ms. The high word contains the desired motor velocity in meters per second and the low word contains the desired motor current as a percentage of the maximum allowed current. Both of these values are sent as IEEE 754 single-precision floating point numbers; therefore, each word has a sign bit, eight exponent bits, and 23 mantissa bits. Both the motor velocity and the motor current are a function of the analog-to-digital (ADC) conversion value from the accelerator potentiometer. The DC_POWER message is shown in Fig. E3. Its ID on the CAN bus is 0x502 and the message is sent over the CAN bus every 100 ms. The high word is reserved and does not contain any value. The low word contains the bus current, which is described in Fig. E3. The bus current always has a fixed value of 1.0, which is converted to IEEE 754 single-precision floating point format before being sent over the CAN bus.

The DC_SWITCH message is shown in Fig. E4. Its ID on the CAN bus is 0x504 and the message is sent every 100 ms. The high word contains the current switch position from the switch inputs on the DB37 connector. The bitwise layout of these switches is shown in the accompanying table in Fig. E4. The low word contains the switch activity and its bit positions are identical to the high word. A bit in the switch activity word will be high if the corresponding bit in the switch position word has changed from the last CAN frame that was sent. This means that the specific switch on the DB37 has changed state.

Received CAN Messages

The MCP2515 (CAN controller) has two receive buffers that store the contents of a CAN message received from the CAN bus. For a message to be received into a one of the two receive buffers, its ID on the CAN bus must be one that is accepted by the MCP2515's filters and masks.

The filters and masks are software configurable registers that can allow a single CAN ID to be accepted, or a range of CAN IDs. There are two masks that are shared among the six filters. The current configuration of these filters and masks is shown below in Table 3.

Table 3: MCP2515 Masks and Filters

Buffer	Filter	Filter Value	Mask	Mask Value	Accepted IDs
RXB0	RXF0	0x403	RXM0	0xFBE0	0x403, 0x423
RXB0	RXF1	0	RXM0	0xFBE0	-
RXB1	RXF2	0x500	RXM1	0xFC00	0x500 – 0x51F
RXB1	RXF3	0x580	RXM1	0xFC00	0x580 – 0x59F
RXB1	RXF4	0	RXM1	0xFC00	-
RXB1	RXF5	0	RXM1	0xFC00	-

As shown above, filters RXF0 and RXF1 apply to RXB0 and RXF2 through RXF5 apply to RXB1. Likewise, RXM0 applies to RXB0 and RXM1 applies to RXB1. The filter registers of the MCP2515 are broken into two separate registers, RXFnSIDH and RXFnSIDL, for standard identifier high and low. The length of the identifier for the filter is only 11 bits long – SID[10:3] is contained in RXFnSIDH and SID[2:0] is contained in the most 3 significant bits of RXFnSIDL. Therefore, when writing to the RXFnSIDH and RXFnSIDL registers of the MCP2515, the value written would be 0x4030 for RXF0, for example. The mask registers are also broken into two separate registers, RXMnSIDH and RXMnSIDL. The same concept from the filter register is also true for the mask registers – the length of the mask identifier is only 11 bits long. SID[10:3] is contained in RXMnSIDH and SID[2:0] is contained in RXMnSIDL. The values in the mask value column of Table 3 represent the absolute value of the RXMnSIDH and RXMnSIDL registers combined. The most significant byte is written to RXMnSIDH and the least significant byte is written to RXMnSIDL. This is not true of the presentation of the filter values, where only the actual filter value is represented.

The existing driver controller's software already had filters and masks configured, however, these had to be modified to accept new CAN messages with the specific values that were to be presented on the screen. In the addition of a new accepted CAN message, the filter value should be modified first. If preventable, the mask value should not be changed, as this would affect the other filters that use that mask. There are three more filters available to accept more CAN messages. The process of modifying a filter value can be done through somewhat of a trial and error process. The filter value must be compared to the mask value, and a value of one in the mask means that an incoming CAN message ID must have the same value of the corresponding filter bit. For example, RXM0 is applied to RXF0 as follows:

RXM0
HEX F B E 0
BINARY **1111** **1011** **111**0 0000
Where the bolded portion represents the actual mask value
Therefore, the actual mask value is
BINARY 111 1101 1111

RXF0
The actual filter value is
BINARY 100 0000 0011

Comparing the mask and the filter
MASK BINARY 111 1101 1111
FILTER BINARY 100 00**0**0 0011
Where the bolded portion represents the bit that will always be accepted
Therefore, the only two accepted IDs under RXF0 are 0x403 and 0x423

The same process is repeated for the rest of the filters to verify the accepted IDs. The filter and mask truth table is presented in Table 4 below.

Table 4: Filter and Mask Truth Table from the MCP2515 Datasheet (Table 4-2) [2]

Mask Bit	Filter Bit	Message Identifier Bit	Accept or Reject Bit
0	x	x	Accept
1	0	0	Accept
1	0	1	Reject
1	1	0	Reject
1	1	1	Accept

As shown in Table 4, if a filter's corresponding mask bit is zero, then that incoming message ID bit will always be accepted. An example of this was shown above with RXM0 and RXF0. If a filter's corresponding mask bit is one and the filter bit is zero, then the incoming message ID bit must always be zero for that bit to be accepted. Likewise, if a filter bit is one, then the incoming message ID bit must always be one for that bit the be accepted. The CAN messages corresponding to the accepted IDs are shown in Table 5 below.

Table 5: Message IDs and Corresponding CAN Messages

Accepted Values	Messages
0x403, 0x423	MC_VELOCITY from both motor controllers
0x500 – 0x51F	All Driver Controller messages for RTR frames
0x580 – 0x59F	All BPS messages

The accepted value range for the last two rows in Table 5 go beyond the CAN message IDs present on the CAN bus. However, this is not a problem because the MCP2515 will never accept these out-of-range CAN IDs, as their messages are not present on the CAN bus. The extended range of IDs was only present to make configuring the masks and filters easier. The driver controller must be able to accept messages with its own message IDs so that it can respond the remote transmit requests. In such a case, another device on the CAN network will send the driver controller a message in which the ID is equal to one of the four from the driver controller. The

driver controller recognizes this condition and responds with the requested message. The 0x580 – 0x59F accepts all the BPS's CAN messages, as most of them are used. The BPS's messages are shown in the Table 6. The layout of the data for each message corresponds to how the data appears on the actual CAN bus. The high word will appear on the CAN bus first and is the most significant word of the data segment. Likewise, the low word will appear on the CAN bus second, and is the least significant word of the data segment.

The driver controller uses all the BPS's CAN messages except BP_CAN_BASE. While the MCP2515 accepts all the BPS's messages, BP_CAN_BASE will not be used for anything, however, it will be accepted into one of the two receive buffers. This is not a problem because the BP_CAN_BASE message is only sent once every 100 seconds. The BP_VMAX message is used for the maximum cell voltage display measurement and the BP_VMIN message is used for the minimum cell voltage display measurement. The BP_TMAX message is used for the pack temperature display measurement, however, it must be noted that this measurement displays the maximum cell temperature, and not the overall battery temperature. The BP_PCDONE message is only used during the driver controller initialization and does not display anything graphically. The BP_ISH message is used for the state of charge measurement and the net current measurement.

Table 6: Battery Protection System CAN Messages

Identification	CAN ID	Period	Data Type	High Word	Low Word
BP_CAN_BASE	0x580	100 seconds	Uint32, Char[4]	Serial Number	“BPv1” string
BP_VMAX	0x581	10 seconds	Float, Float	Max voltage cell number	Max voltage value (V)
BP_VMIN	0x582	10 seconds	Float, Float	Min voltage cell number	Min voltage value (V)
BP_TMAX	0x583	10 seconds	Float, Float	Max temp cell number	Max temp value (F)
BP_PCDONE	0x584	RTR/PC	Uint32, Char[4]	Serial number	“BPv1” string
BP_ISH	0x585	10 seconds	Float, Float	SOC (mA/sec)	Current (mA)

Testing and Verification

Overview

Testing the DDC PCB in the current version of the solar car was not possible at the time of writing this report. Therefore, the completed PCB was tested using a CAN test bench and an Analog Discovery 2. The CAN test bench consists of a small dongle that plugs into any computer's USB port. The other end of the dongle contains a DB9 connector that is meant to be connected to the CAN bus. The dongle contains a Phillips SJA1000 CAN Controller and a Phillips 82C251 CAN Transceiver. Therefore, no other CAN devices were required to be connected to the CAN bus other than the DDC, as a CAN bus requires at least two devices on the network for full functionality. The CANUSB dongle was purchased from canusb.com, and along with a piece of software to send and receive CAN messages over USB, the test bench setup was complete. The software was borrowed from the solar car team and required an installation of Windows XP, which was done in a virtual machine, for correct function of the software.

MSP430 Clock Frequencies

To verify that the MSP430's clocking system was configured correctly, the corresponding test points on the PCB were probed using an Analog Discovery 2. Table 2 shows the expected clock frequencies of the three different clocks. Fig. 27 shows that the frequency of the measured ALCK signal was 32.768 kHz, which matches the configured frequency from Table 2.

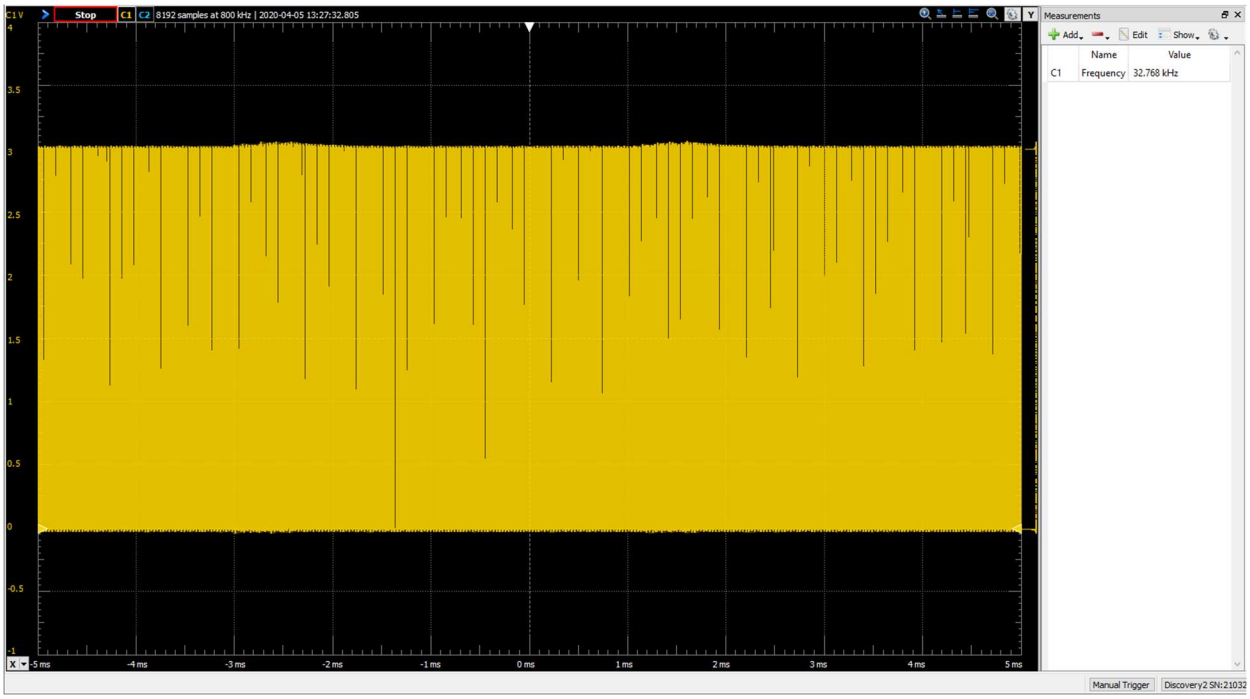


Figure 27: External ACLK output of the MSP430

The measured MCLK signal and its corresponding frequency is shown in Fig. 28 below. The measured frequency matches the configured frequency from Table 2.

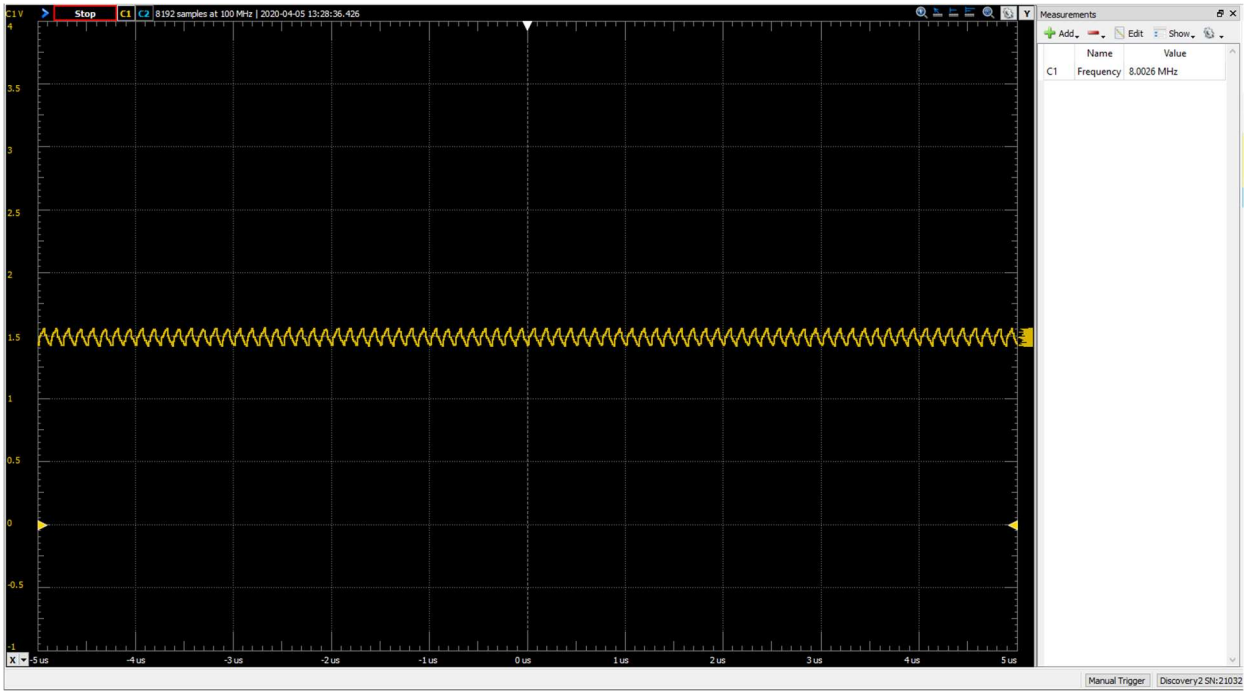


Figure 28: External MCLK output of the MSP430

The measured SMCLK signal and its corresponding frequency is shown in Fig. 29 below. The measured frequency matches the configured frequency from Table 2.

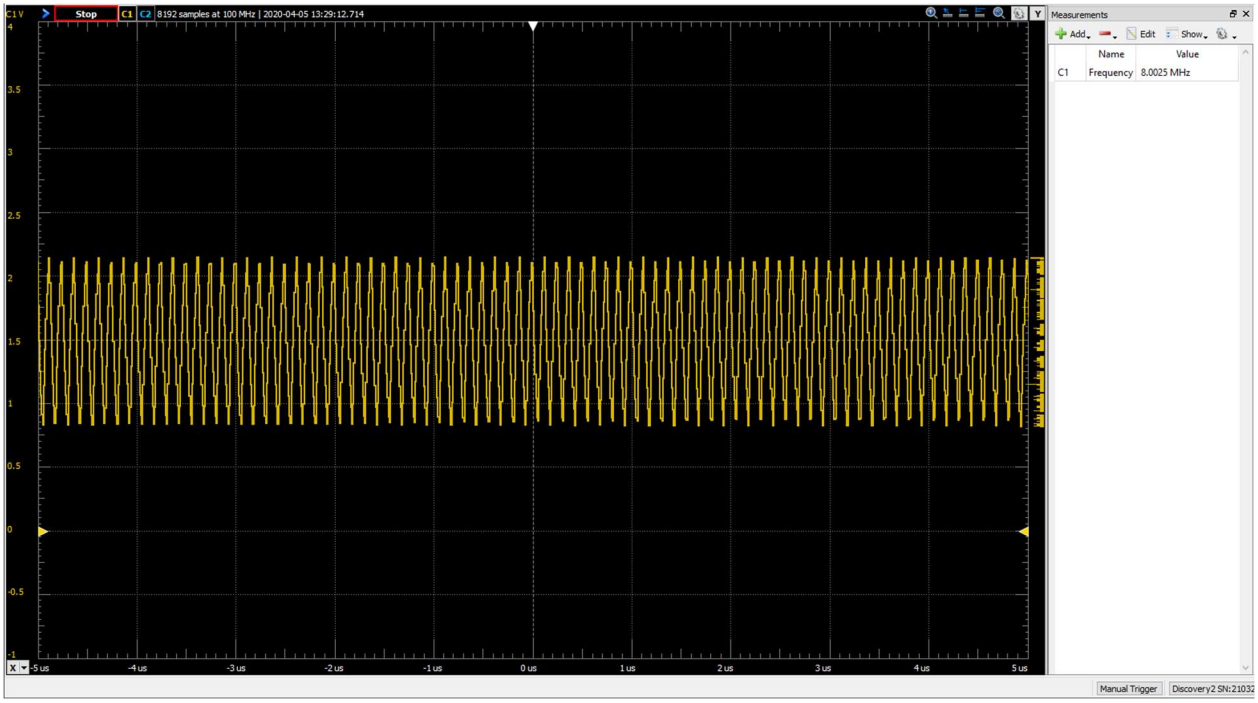


Figure 29: External SMCLK output of the MSP430

Digital Inputs

To verify the correct function of the 18 different digital inputs, the DC_SWITCH CAN message was used. The high word of the DC_SWITCH message contains the current switch position on the DB37 connector. Therefore, to test the digital input protection circuitry, the driver controller code, and the correct function of the driver controller CAN messages, all 18 digital inputs on the DB37 connector were tested both in the ON and OFF state. However, for the purpose of this report, only one digital input, BRAKE_1, will be presented. The CANUSB software contains a terminal window which prints out all the messages on the CAN bus. Therefore, the message with an ID of 0x504 was found and the data segment was the message was analyzed. For the first test case, the switch for the BRAKE_1 input was left open on the

DB37 connector. Because the digital protection circuit has a pull-up resistor to the 12V rail, the MOSFET will be on, which will pull the input to the MSP430 low. An example of this scenario is explained with Fig. 6. Therefore, we can expect the BRAKE_1 bit in the DC_SWITCH CAN message to be low as well. This CAN message, as received by the CANUSB software, is as follows:

```
id = 00000504 timestamp = 000B4800 len = 8 data =00 00 00 00 00 00 00 00
```

All values of the CAN message are presented in hexadecimal. As seen from the data segment of the message, the high word consists of all zeros, which is what was expected. Next, the BRAKE_1 switch was closed. In this scenario, the input to the MSP430 should be high, and therefore, the BRAKE_1 bit of the DC_SWITCH message should also be high. The corresponding CAN message is as follows:

```
id = 00000504 timestamp = 0014C988 len = 8 data =10 00 00 00 00 00 00 00
```

As seen from the data segment, the most significant byte of the high word is 0x10. If this is compared to the bit layout in Fig. E4, then it can be confirmed that the BRAKE_1 bit is high in the DC_SWITCH message. This same process was repeated for the 17 other digital inputs to verify the correct function of both the physical hardware and the software. The two analog inputs and the two analog outputs were tested with a potentiometer to simulate the correct function of the accelerator. This also proved to be successful.

Display Measurements

Perhaps the most important testing that was done was testing the measurements on the display. The measurements were tested by sending the associated CAN message through the CANUSB software, which would be sent from the motor controllers and the BPS. Only the active measurements were tested, which excluded the state of charge and the pack voltage measurements. The active measurement values were sent in IEEE 754 single-precision floating

point format, which is how the motor controllers and the BPS send the values for these measurements. For the speed measurement, an ID of 0x403 was set in the CANUSB software and a speed of 10 meters/second was set. Using an IEEE 754 floating point converter, a hexadecimal value of 0x41200000 was obtained. Because the driver controller software sends data to the CAN bus least-significant byte first within each word, messages are also expected to be received in this manner as well. Therefore, the data was inputted to the CANUSB software as shown in Fig. 30.

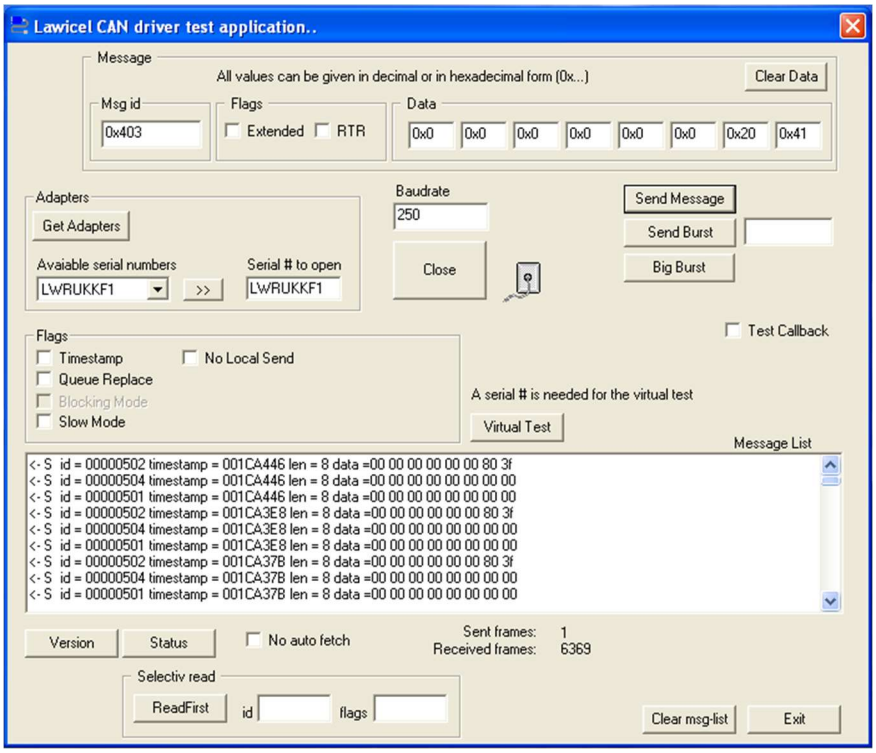


Figure 30: A motor velocity of 10 m/s sent to the DDC

Because the motor velocity in meters/second is the low word of the data segment of the CAN message, 0x41200000 was set in the low word of the data to be sent. Notice that the most significant byte of 0x41200000 is set in the least significant byte's position. Once the message was sent to the CAN bus, the speed value was updated on the display as shown in Fig. 31 below.

Figure 31: Updated speed value from the CAN bus

This is the correct value to be displayed on the screen because 10 meters/second converted to miles per hour is 22.3694, which the display rounds down to 22. Therefore, the correct function of the speed measurement has been proven.

To test the maximum cell voltage measurement, an ID ox 0x581 was set in the CANUSB software. A value of 3.5V was chosen to represent the maximum cell voltage, which is not necessarily representative of the actual solar car's maximum cell voltage. After converting 3.5V to IEEE 754 format, the obtained hexadecimal value was 0x40600000. The ID and data were set in the CANUSB software as shown in Fig. 32 below.

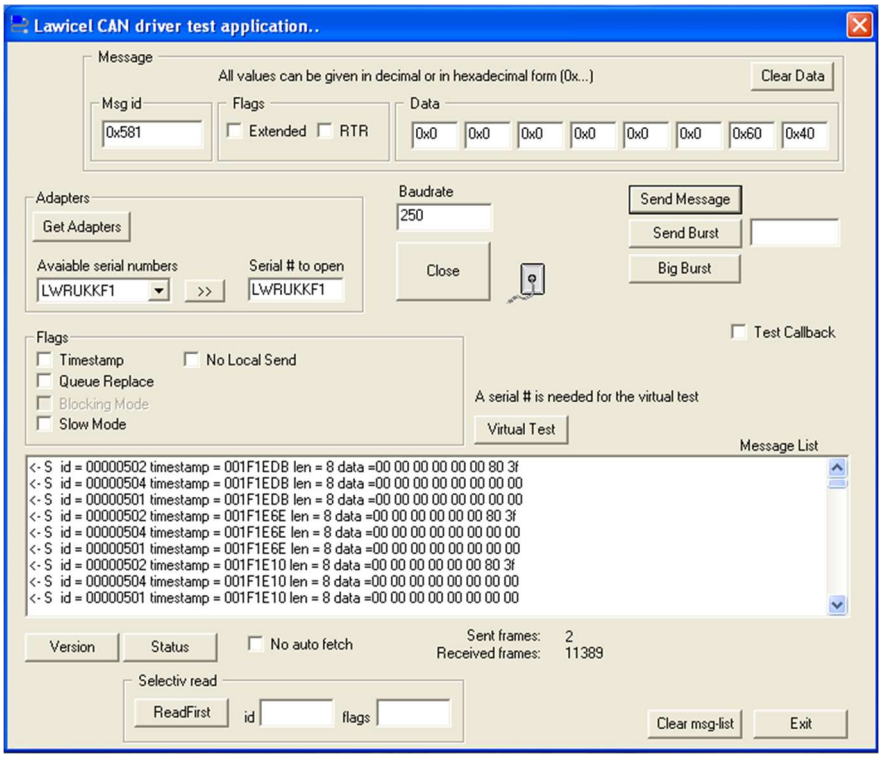


Figure 32: A maximum cell voltage of 3.5V sent to the DDC

The maximum cell voltage value was set in the least significant word of the data segment, which is how the BPS would send the message. After transmitting the message to the CAN bus, the maximum cell voltage measurement on the display was updated as shown in Fig. 33 below.

Figure 33: Updated maximum cell voltage value from the CAN bus

Therefore, the correct function of the maximum cell voltage measurement has been proven.

To test the minimum cell voltage measurement, an ID ox 0x582 was set in the CANUSB software. A value of 2.5V was chosen to represent the minimum cell voltage, which is not necessarily representative of the actual solar car's minimum cell voltage. After converting 2.5V to IEEE 754 format, the obtained hexadecimal value was 0x40200000. The ID and data were set in the CANUSB software as shown in Fig. 34 below.

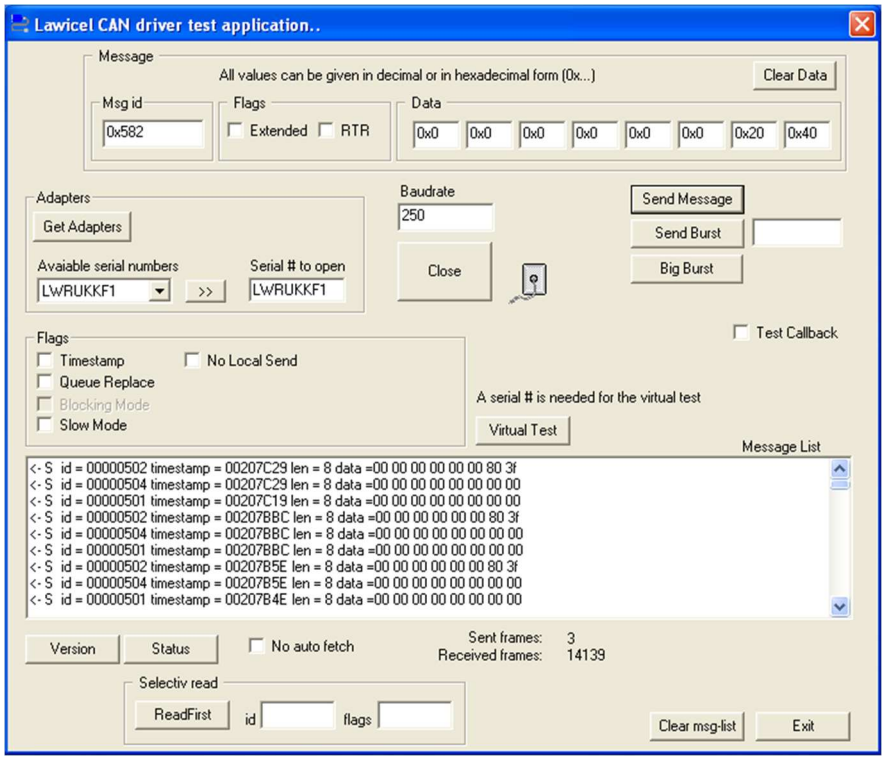


Figure 34: A minimum cell voltage of 2.5V sent to the DDC

The minimum cell voltage value was set in the least significant word of the data segment, which is how the BPS would send the message. After transmitting the message to the CAN bus, the minimum cell voltage measurement on the display was updated as shown in Fig. 35 below.

Figure 35: Updated minimum cell voltage from the CAN bus

Therefore, the correct function of the maximum cell voltage measurement has been proven.

To test the pack temperature measurement - which is really the maximum cell temperature - an ID ox 0x583 was set in the CANUSB software. A value of 90 F was chosen to represent the maximum cell temperature, which is not necessarily representative of an actual value from the solar car. After converting 90 F to IEEE 754 format, the obtained hexadecimal value was 0x42B40000. The ID and data were set in the CANUSB software as shown in Fig. 36 below.

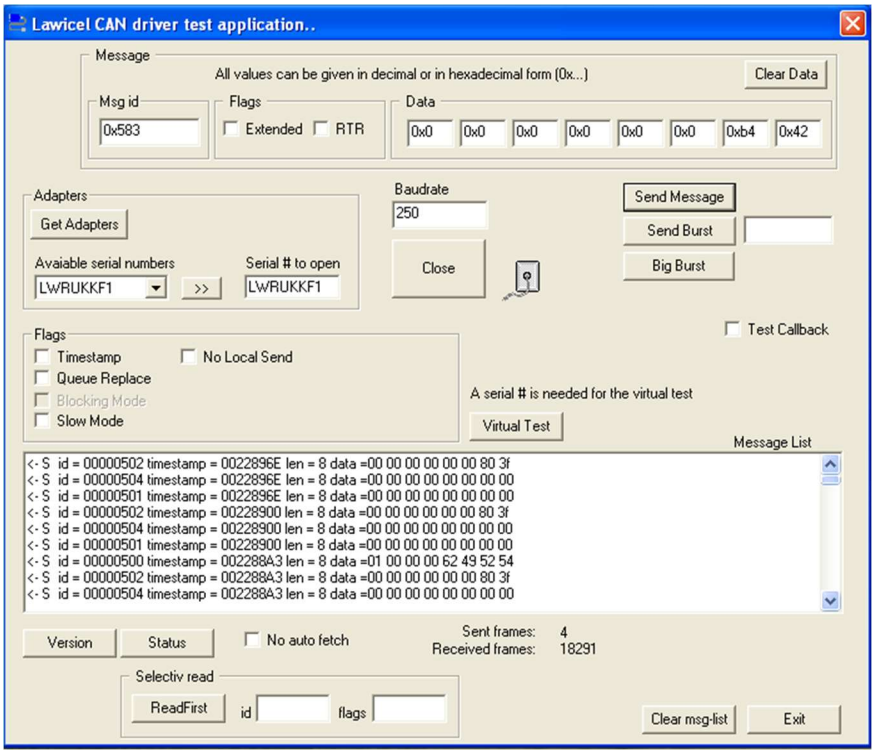


Figure 36: A maximum cell temperature of 90F sent to the DDC

The maximum cell temperature value was set in the least significant word of the data segment, which is how the BPS would send the message. After transmitting the message to the CAN bus, the maximum cell temperature measurement on the display was updated as shown in Fig. 37 below.

Figure 37: Updated maximum cell voltage from the CAN bus

The maximum cell temperature appears in green text because it is under the current threshold of 100 F. The value was changed to 110 F and retransmitted to the CAN bus, which is expected to read 110 F in red text, as this value is above the current temperature threshold. This is shown in Fig. 38 below.

Figure 38: Updated maximum cell voltage from the CAN bus

Therefore, the correct function of the pack temperature measurement has been proven.

To test the net current measurement, an ID of 0x585 was set in the CANUSB software. A value of 1A was chosen to represent the net current, which would be sent by the BPS as 1000 mA. After converting 1000 mA to IEEE 754 format, the obtained hexadecimal value was 0x447A0000. The ID and data were set in the CANUSB software as shown in Fig. 39 below.

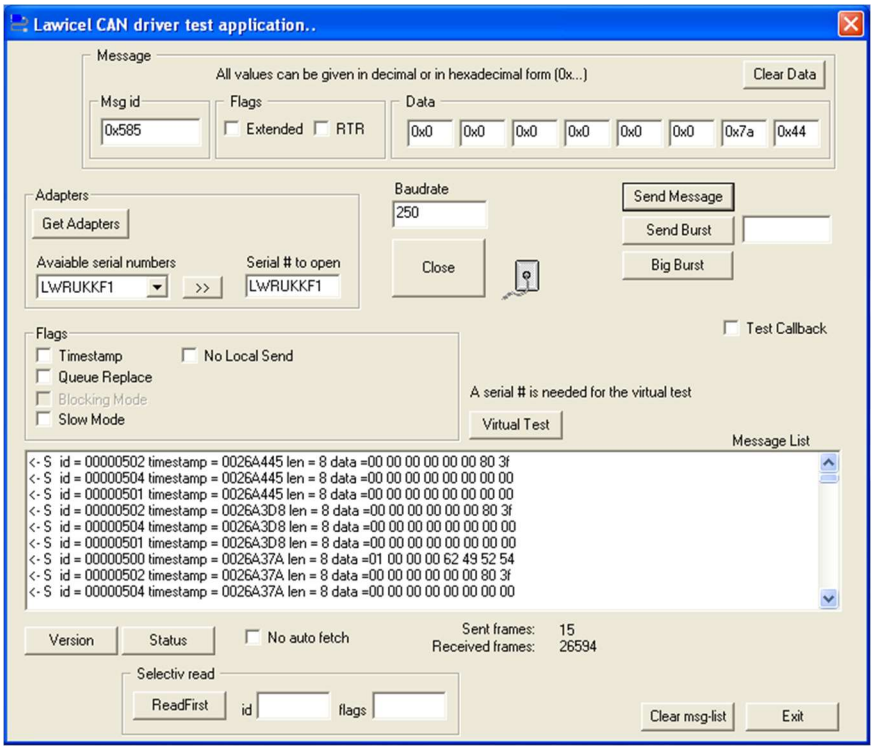


Figure 39: A net current value of 1A sent to the DDC

The net current value was set in the least significant word of the data segment, which is how the BPS would send the message. After transmitting the message to the CAN bus, the net current measurement on the display was updated as shown in Fig. 40 below.

Figure 40: Updated net current from the CAN bus

The net current measurement appears in green text because its value is positive, which indicates that a net current of 1A is going into the battery. The value was changed to -1A and retransmitted on the CAN bus, which is expected to be shown in red because it represents a net current of 1A coming out of the battery. This is shown in Fig. 41 below.

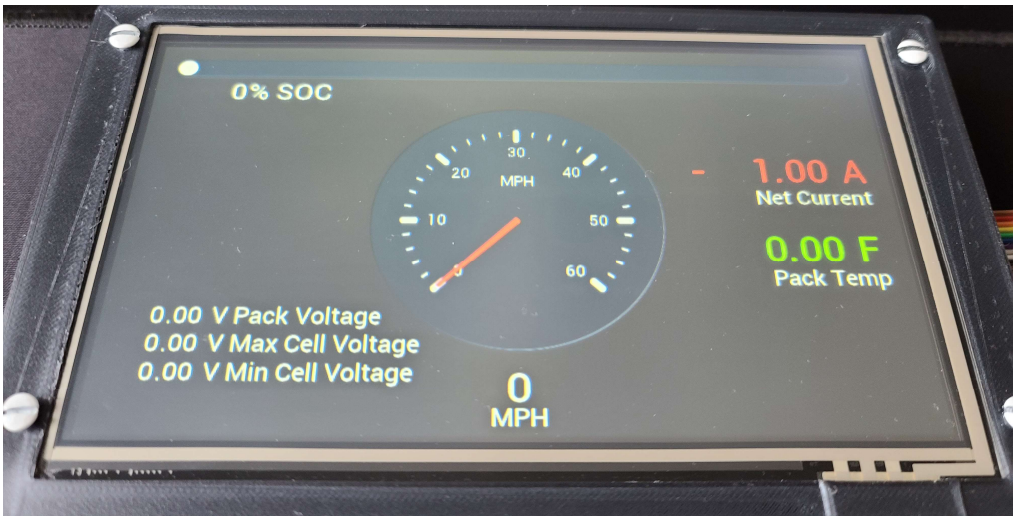


Figure 41: Updated net current from the CAN bus

Therefore, the correct function of the net current measurement has been proven.

The turn signals are the only graphical elements on the screen that are not currently controlled by a CAN message, they are controlled instead by digital inputs. The turn signals and hazard were tested and when activated, appeared on the display correctly.

Conclusion

Evaluation of Specifications

The following subsections explain how the characteristics of the finished Display and Driver Controller PCB compare to the proposed characteristics in three categories: physical characteristics, design, and functionality.

Physical Characteristics

Sizing

1. Display must have appropriate sizing to allow it to be mounted within the cockpit. This should be anywhere from 5”- 10.”

   The 7-inch display is housed in a 3D printed enclosure that is 8.74 inches diagonally. Therefore, this specification was met.

2. There must be an overall decrease in size from the previous driver controller box and display controller box. There will be a reduction from two boxes and two PCBs to one box and one PCB.

   The completed design consists of a single PCB and its enclosure that measures 9.85 inches by 6.85 inches. Therefore, the specification was met.

Cosmetic

1. This display will be able to show more information than before and in a clearer and more concise way.

   The new display shows all necessary information on a 7-inch color LCD screen. The information displayed on the previous design was smaller, unclear, and presented on four different screens. Therefore, this specification was met.

2. Mounting case must be designed and 3D printed to mount the display inside the solar car's cockpit.

   The enclosure for the display was 3D printed and can be mounted inside the solar car's cockpit. Therefore, this specification was met.

3. PCB must be housed in a box in a similar fashion as other main components.

   The PCB is housed in a 3D printed box that can be mounted in the solar car. Therefore, this specification was met.

Design

Microcontroller

1. Display and Driver Control Unit must use a low power microcontroller.

   The DDC uses an MSP430F5438A, an ultra-low power microcontroller. Therefore, this specification was met.

Power Rating

2. There is not a hard number that can be exceeded as far as power consumption for the combination of the PCB and Display, but the whole system should be in the range of a few watts *at most*. Preferably, the power consumption would be less than 1W.

   The whole system uses roughly 1.25W when the display's backlight is configured at 15% brightness. At this brightness, the display is still visible and easily readable. Therefore, this specification was met.

Functionality

User Interface of Display Unit

1. Display must be programmed to display results in a friendly interface that can easily be interpreted by the driver of the vehicle. Display should feature a functional speedometer, state of charge, min/max battery cell voltage, pack voltage, net current of the battery, and temperature of the battery, amongst other non-critical information such as time, turn signal indicators, etc.

   The display presents all measurements in an intuitive manner. These measurements include speed (presented in both analog and digital), state of charge, minimum and maximum cell voltage, pack voltage, net current, pack temperature, turn signal indicators, and hazard indicators. Therefore, this specification was met.

Driver Controller

1. Send CAN messages to motor controller based on acceleration pedal potentiometer reading.
2. Send CAN messages to light board to control turn signals and other LEDs based on user input.
3. Send CAN messages to BPS board based on enable signal during car startup.

   The microcontroller communicates respective measurements on the CAN bus with the motor controller, light board, and BPS board. Therefore, this specification was met.

Connectivity

1. Display unit must interface using SPI with a microcontroller to receive accurate measurements of speed, state of charge, etc. The microcontroller will interface with the CAN network to get all the information to display and then control the display with that information.

   The microcontroller communicates to the display's controller via SPI, which communicates directly with the panel. Therefore, this specification was met.

2. Driver Controller and Display Driver must be consolidated into one PCB and maintain all pre-existing functionality, while adding additional functionality. This PCB will have GPIO, CAN, and SPI connectivity.

   The new DDC consolidated the previous driver controller and display driver into one PCB while maintaining all functionality. Additional functionality was implemented by adding more meaningful measurements to the display. The PCB has general-purpose-input-ouput (GPIO) pins, CAN, and SPI connectivity. Therefore, this specification was met.

Recommendations

Several recommendations have been prepared for future groups that partake in a follow-up project, or anyone who decides to reproduce this project. There would be some merit to using an STM32 microprocessor instead of an MSP430. While an MSP430 provides a more immersive hardware engineering experience and ideal specifications for this application, using an STM32 microprocessor would save a significant amount of time in programming. This would leave more time for improvements in other areas of the project. STM32 microprocessors can be programmed easily using STMCubeMX, which is a tool that does not currently exist for the MSP430 family of microprocessors. Using an STM32 microprocessor is only a loose recommendation as it would not improve any functionality of the project but would allow more time to be invested elsewhere.

In the reproduction of the hardware components of the project, it is recommended to decrease the overall size of the PCB. This project produced a first-generation product, and the size of the PCB was not as much of a consideration as the overall functionality. Therefore, this is something that should be considered in the future. Furthermore, while 3D printing technology allowed the design of the PCB to occur first, this is something that should also be considered in the next generation of the DDC. It took three 3D prints to finalize the enclosure for the PCB because of tolerance issues, as there was little mechanical experience among team members. Given a team with little 3D printing experience and resources, it is recommended to design the PCB to fit an existing enclosure with tolerances in mind.

References

[1] Bridgetek, "FT81X Series Programmers Guide," FT812 datasheet, Oct. 2018.

[2] Microchip Technology, "Stand-Alone CAN Controller with SPI Interface," MCP2515 datasheet, Aug. 2018.

Appendix A

EVE_CLEAR_COLOR_RGB
Found on page 121 of FT812 Programmer Guide
The encoding table is wrong in the datasheet!

Encoding

31 24	23 16	15 8	7 0
0x02	Red	Green	Blue

Parameters

Red – The Red value used when the color buffer is cleared.
Green – The Green value used when the color buffer is cleared.
Blue – The Blue value used when the color buffer is cleared.

Description

Sets the color values used by a following EVE_CLEAR command.

EVE_CLEAR
Found on page 118 of FT812 Programmer Guide

Encoding

31 24	23 3	2	1	0
0x26	Reserved	C	S	T

Parameters

C - Clear color buffer. Setting this bit to 1 will clear the color buffer of the FT81X to the preset value. Setting this bit to 0 will maintain the color buffer of the FT81X with an unchanged value.
S - Clear stencil buffer. Setting this bit to 1 will clear the stencil buffer of the FT81X to the preset value. Setting this bit to 0 will maintain the stencil buffer of the FT81X with an unchanged value.
T - Clear tag buffer. Setting this bit to 1 will clear the tag buffer of the FT81X to the preset value. Setting this bit to 0 will maintain the tag buffer of the FT81X with an unchanged value.

Description

If all three buffers are cleared at the same time, this command will change the background to the color specified by the function EVE_CLEAR_COLOR_RGB.

Figure A1: CLEAR_COLOR_RGB and CLEAR commands

EVE_CMD_GAUGE
Found on page 187 of FT812 Programmer Guide

Encoding

Offset	Parameter	Length
+0	CMD_GAUGE (0xFFFFFF13)	4 bytes
+4	X	2 bytes
+6	Y	2 bytes
+8	R	2 bytes
+10	Options	2 bytes
+12	Major	2 bytes
+14	Minor	2 bytes
+16	Value	2 bytes
+18	Range	2 bytes

Parameters

X - X-coordinate of gauge center, in pixels
Y - Y-coordinate of gauge center, in pixels
R - Radius of the gauge, in pixels
Options - By default the gauge dial is drawn with a 3D effect and the value of options is zero.
OPT_FLAT removes the 3D effect. With option OPT_NOBACK, the background is not drawn. With option OPT_NOTICKS, the tick marks are not drawn. With option OPT_NOPOINTER, the pointer is not drawn.
Major - Number of major subdivisions on the dial, 1-10
Minor - Number of minor subdivisions on the dial, 1-10
Value - Gauge indicated value, between 0 and range, inclusive
Range - Maximum value

Description

Draws a gauge widget. The total length of the command is 20 bytes.

Figure A2: GAUGE command

EVE_CMD_TEXT
Found on page 213 of FT812 Programmer Guide

Encoding

Offset	Parameter	Length
+0	CMD_TEXT (0xFFFFFF0C)	4 bytes
+4	X	2 bytes
+6	Y	2 bytes
+8	Font	2 bytes
+10	Options	2 bytes
+12	S	1 byte
..	..	1 byte
..	0 (null character to terminate string)	1 byte

Parameters

X - x-coordinate of text base, in pixels
Y - y-coordinate of text base, in pixels
Font - font to use for text, 0-31. See ROM and RAM Fonts
Options - By default (x,y) is the top-left pixel of the text and the value of options is zero.
OPT_CENTERX centers the text horizontally, OPT_CENTERY centers it vertically.
OPT_CENTER centers the text in both directions.
OPT_RIGHTX right-justifies the text, so that the x is the rightmost pixel.
Text - The text string itself

Description

This is a variable length command that draws text on the screen. Because the display is designed to read ASCII characters, each character is a byte. Furthermore, because each graphics command must be 4-byte aligned, the very minimum length of this command is 16 bytes. The C function always rounds up to the nearest 4-byte multiple. For example, if the string was “Hi”, the C function would write ‘H’ at +12, ‘i’ at +13, 0x00 at +14 for the null termination character, and 0x00 at +15 to pad the length to 16 bytes.

Figure A3: TEXT command

EVE_COLOR_RGB
Found on page 126 of FT812 Programmer Guide
The encoding table in the datasheet is wrong!

Encoding

31 24	23 16	15 8	7 0
0x04	Red	Green	Blue

Parameters

Red – The Red value for the current color.
Green – The Green value for the current color.
Blue – The Blue value for the current color.

Description

Sets the color values to be used for the following draw operation. This function is NOT the same as EVE_CLEAR_COLOR_RGB, which only applies to an EVE_CLEAR command. This function applies to all draw commands.

EVE_CMD_NUMBER
Found on page 218 of FT812 Programmer Guide

Encoding

Offset	Parameter	Length
+0	CMD_NUMBER (0xFFFFFF2E)	4 bytes
+4	X	2 bytes
+6	Y	2 bytes
+8	Font	2 bytes
+10	Options	2 bytes
+12	n	4 bytes

Parameters

X - x-coordinate of text base, in pixels
Y - y-coordinate of text base, in pixels
Font - font to use for text, 0-31. See ROM and RAM Fonts
Options - By default (x,y) is the top-left pixel of the text. OPT_CENTERX centers the text horizontally, OPT_CENTERY centers it vertically. OPT_CENTER centers the text in both directions. OPT_RIGHTX right-justifies the text, so that the x is the rightmost pixel. By default the number is displayed with no leading zeroes, but if a width 1-9 is specified in the options, then the number is padded if necessary with leading zeroes so that it has the given width. If OPT_SIGNED is given, the number is treated as signed, and prefixed by a minus sign if negative.
number - The number to display, is either unsigned or signed 32-bit, in the base specified in the preceding CMD_SETBASE. If no CMD_SETBASE appears before CMD_NUMBER, it will be in decimal base.

Description

Draws a number on the screen. The total length of the command is always 16 bytes.

Figure A4: COLOR_RGB and NUMBER commands

EVE_CMD_PROGRESS
Found on page 200 of FT812 Programmer Guide

Encoding

Offset	Parameter	Length
+0	CMD_PROGRESS (0xFFFFFF0F)	4 bytes
+4	X	2 bytes
+6	Y	2 bytes
+8	W	2 bytes
+10	H	2 bytes
+12	Options	2 bytes
+14	Value	2 bytes
+16	Range	2 bytes
+18	Padding	2 bytes

Parameters

X - x-coordinate of progress bar top-left, in pixels
Y - y-coordinate of progress bar top-left, in pixels
W - width of progress bar, in pixels
H - height of progress bar, in pixels
Options - By default the progress bar is drawn with a 3D effect and the value of options is zero.
Options OPT_FLAT remove the 3D effect and its value is 256
Value - Displayed value of progress bar, between 0 and range inclusive
Range - Maximum value

Description

Draws a progress bar on the screen. The total length of the command is 20 bytes because 2 bytes must be added to keep the command 4-byte aligned.

Figure A5: PROGRESS command

EVE_VERTEX
Found on page 146 of FT812 Programmer Guide

Encoding

31 30	29 21	20 12	11 7	6 0
0x2	X	Y	handle	cell

Parameters

X - x-coordinate in pixels.
Y - y-coordinate in pixels.
Handle - Bitmap handle. The valid range is from 0 to 31.
C - Cell number. Cell number is the index of the bitmap with same bitmap layout and format. For example, for handle 31, the cell 65 means the character "A" in built in font 31.

Description

Draws a character on the screen for a specific font. The characters use their ASCII equivalent code. EVE_BEGIN and EVE_END must be called before and after this command, respectively, for it to work properly.

Figure A6: VERTEX command

Appendix B

Host Memory Write Transaction

- To start a memory write command, the two most significant bits of the address must be 2'b01. Then, send the 22-bit address starting with the MSB.
- Multiple bytes can be written in one transaction, because the address pointer of the FT812 auto-increments to the very next memory location.
- Send the LSB of the data first, ending with the MSB of the data.

Corresponding C Function	7	6	5	4	3	2	1	0	
spi_transmitEVE	1	0	Address[21:16]						Write Address
spi_transmitEVE	Address[15:8]								
spi_transmitEVE	Address[7:0]								
spi_transmitEVE	Byte 0 (Least Significant Byte)								Write Data
spi_transmitEVE	Byte n (Most Significant Byte)								

Figure B1: Host Memory Write Transaction Template

Host Memory Read Transaction

- To start a memory read command, the two most significant bits of the address must be 2'b00. Then, send the 22-bit address starting with the MSB.
- Next, a dummy byte must be sent so the FT812 has time to put the contents of that memory location in its transmit buffer
- Lastly, exchange a byte with the FT812 to receive a byte in return. The FT812 always sends the LSB of the data first, ending with the MSB of the data.
- The address pointer of the FT812 will auto-increment to the next memory location

Corresponding C Function	7	6	5	4	3	2	1	0	
spi_transmitEVE	0	0	Address[21:16]						Read Address
spi_transmitEVE	Address[15:8]								
spi_transmitEVE	Address[7:0]								
spi_transmitEVE	Dummy Byte								
spi_exchangeEVE	Byte 0 (Least Significant Byte)								Read Data
spi_exchangeEVE	Byte n (Most Significant Byte)								

Figure B2: Host Memory Read Transaction Template

Host Command

- There are a number of hosts commands listed in Table 4-5 of the FT81x data sheet.
- The only host commands used in the driver controller activate the FT812 and select the internal clock.
- Table 4-5 shows exactly what values to send for all 3 bytes.

Corresponding C Function	7	6	5	4	3	2	1	0	
spi_transmitEVE	0	1	Command[5:0]						1st byte
spi_transmitEVE	Parameter for the command								2nd byte
spi_transmitEVE	0	0	0	0	0	0	0	0	3rd byte

Figure B3: Host Command Template

Appendix C

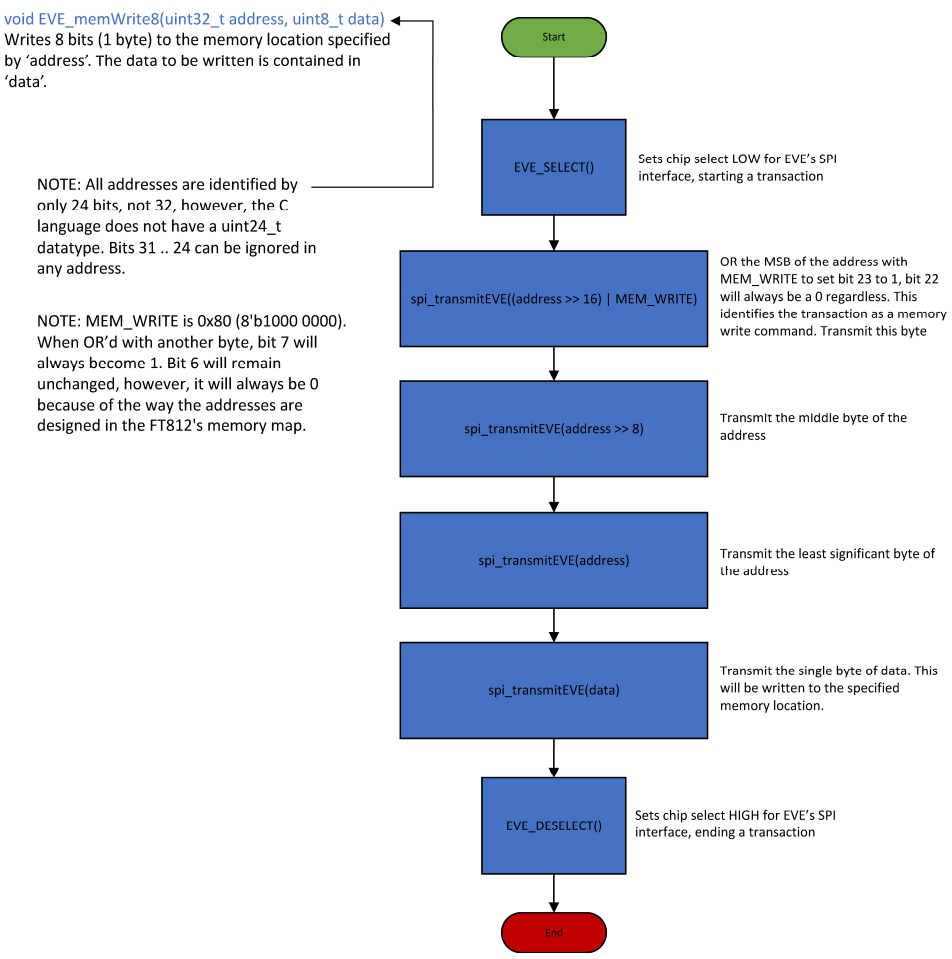


Figure C1: Memory Write 8 Bits Function Flowchart

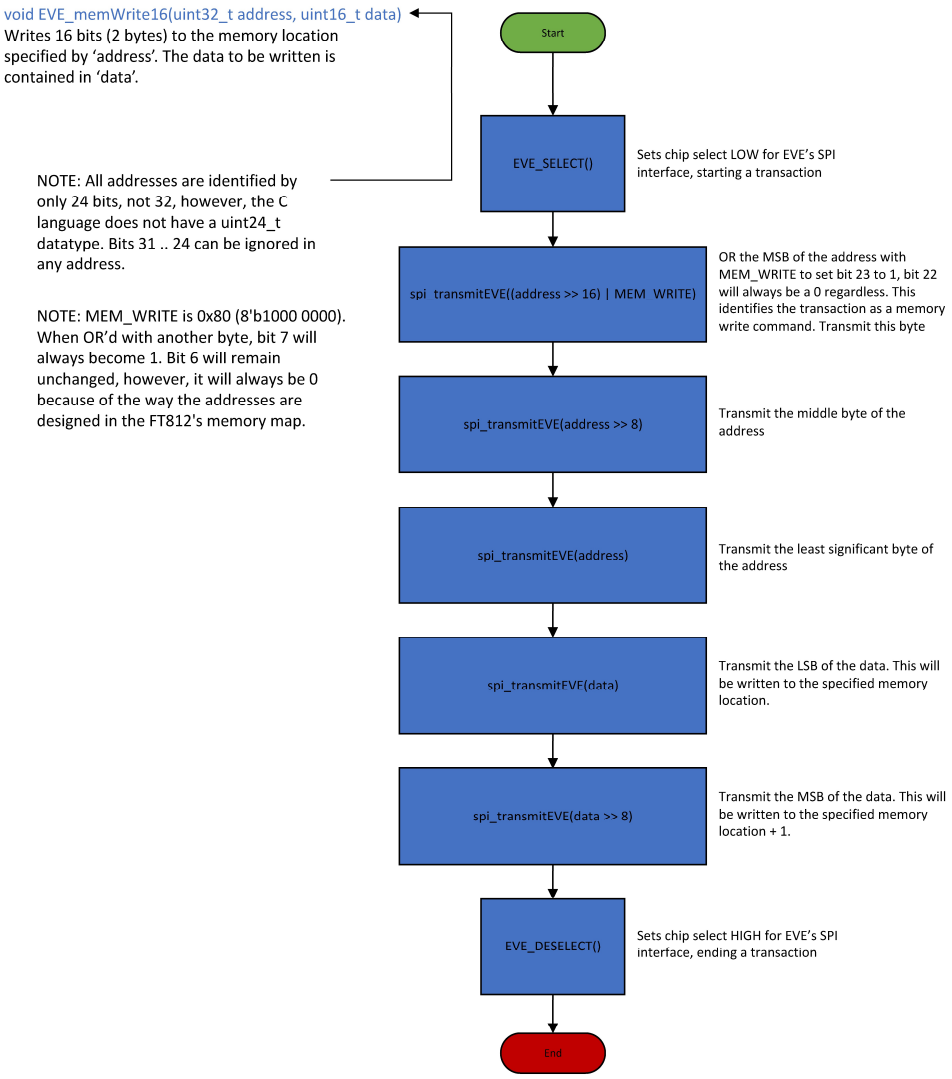


Figure C2: Memory Write 16 Bits Function Flowchart

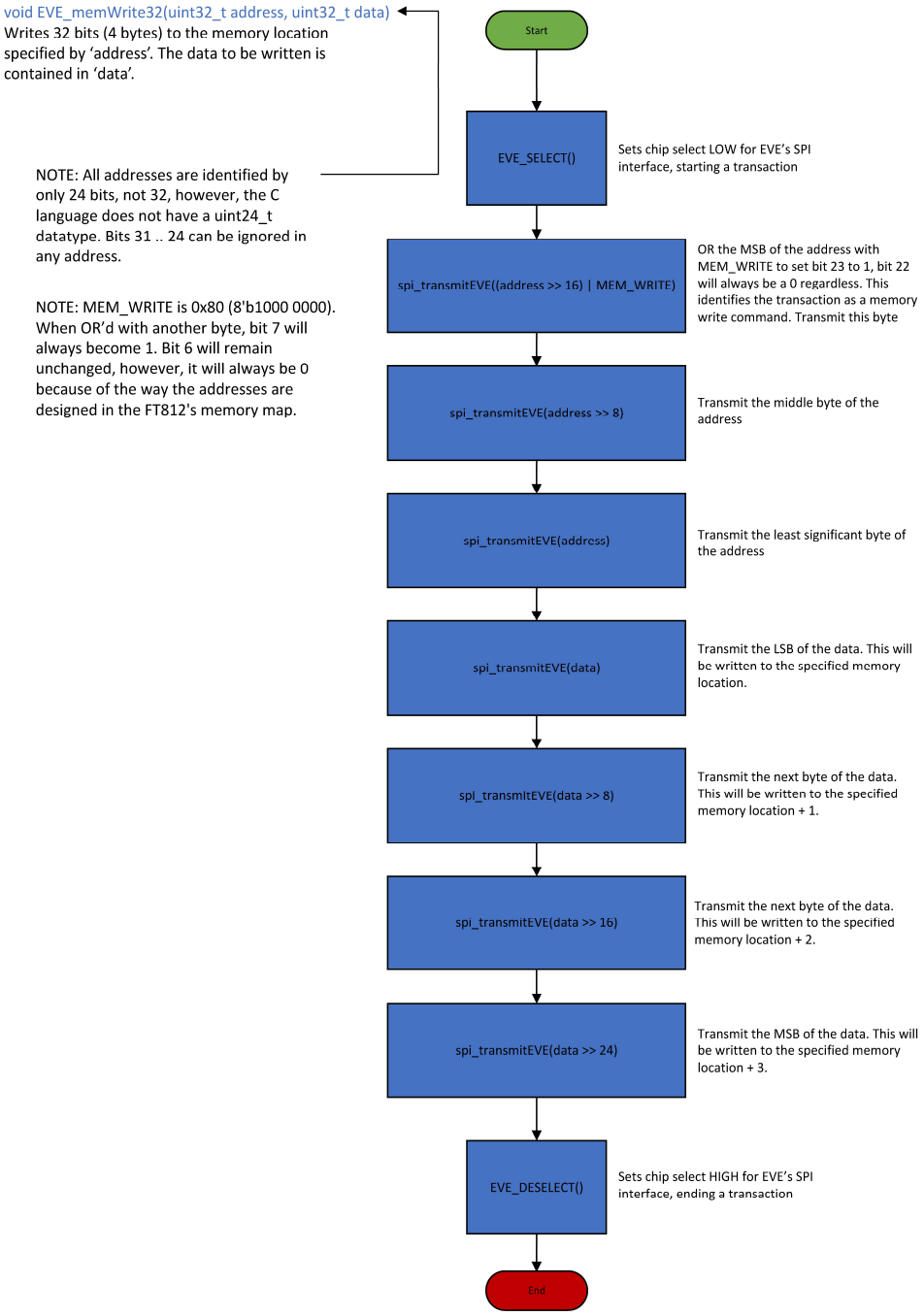


Figure C3: Memory Write 32 Bits Function Flowchart

char EVE_memRead8(uint32_t address)
Reads 8 bits (1 byte) from the memory location specified by 'address'. Returns the contents of 'address' in a char variable.

NOTE: All addresses are identified by only 24 bits, not 32, however, the C language does not have a uint24_t datatype. Bits 31 .. 24 can be ignored in any address.

NOTE: MEM_READ is 0x3F (8'b0011 1111). When ANDed with another byte, the most significant 2 bits will always be 0 and the lower 6 bits will remain unchanged.

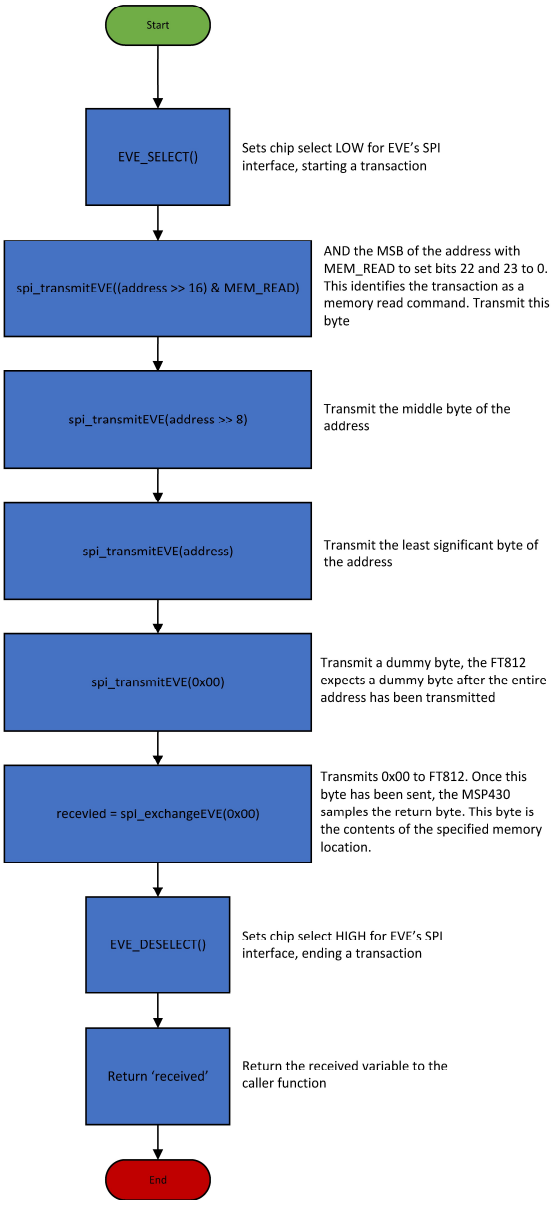


Figure C4: Memory Read 8 Bits Function Flowchart

uint16_t EVE_memRead16(uint32_t address)
Reads 16 bits (2 bytes) from the memory location specified by 'address'. Returns the contents of 'address' in a uint16_t variable.

NOTE: All addresses are identified by only 24 bits, not 32, however, the C language does not have a uint24_t datatype. Bits 31 .. 24 can be ignored in any address.

NOTE: MEM_READ is 0x3F (8'b0011 1111). When ANDed with another byte, the most significant 2 bits will always be 0 and the lower 6 bits will remain unchanged.

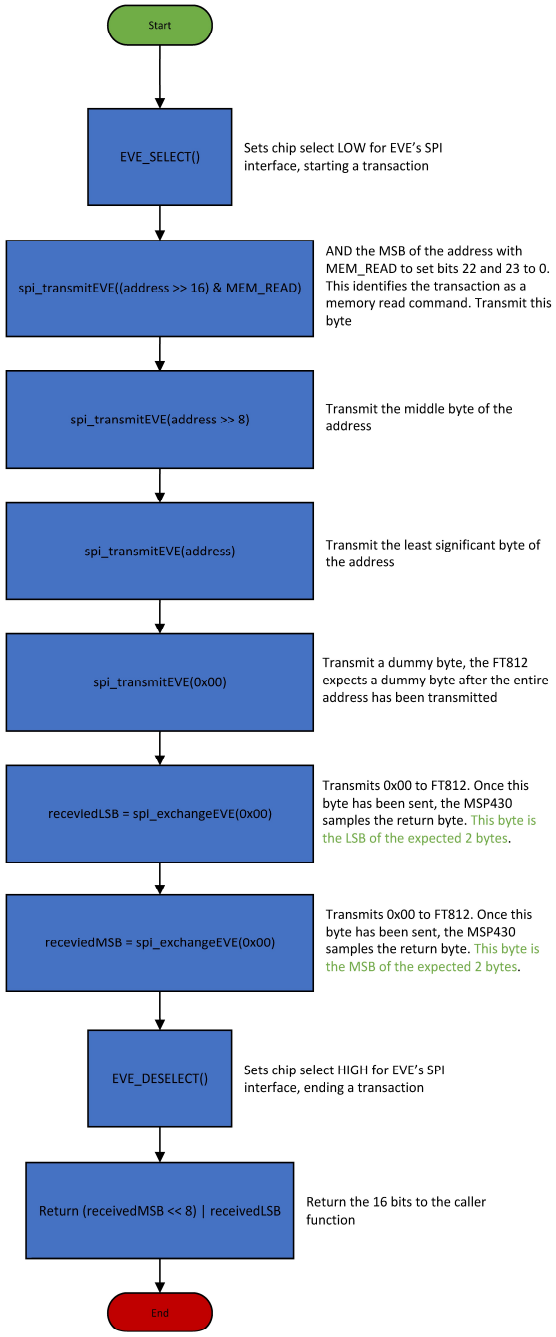


Figure C5: Memory Read 16 Bits Function Flowchart

Appendix D

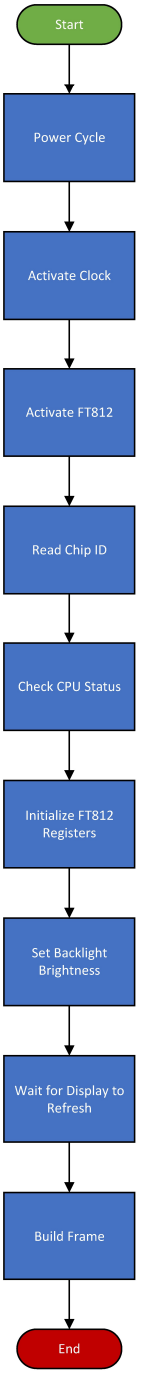


Figure D1: Display Initialization Flowchart

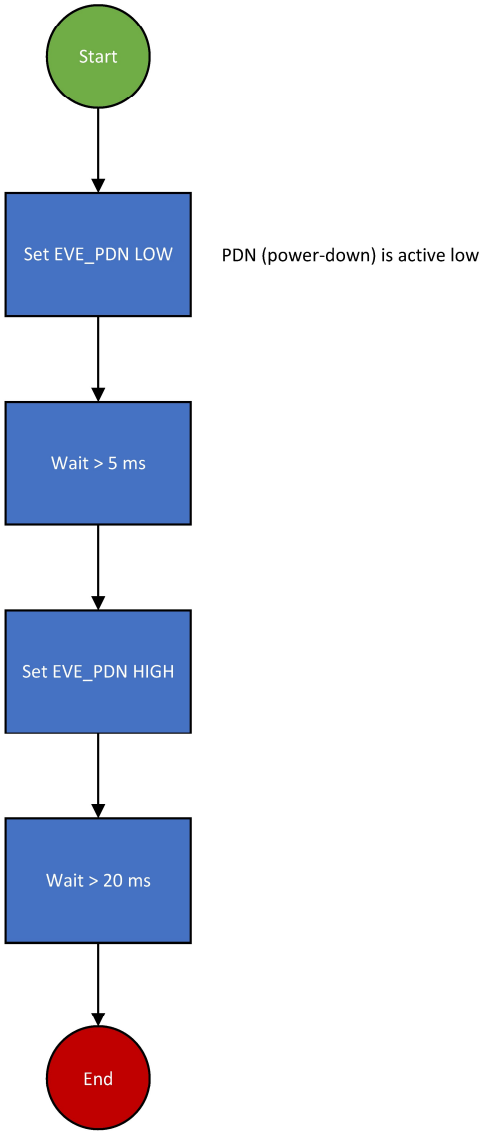


Figure D2: FT812 Power Down Sequence Flowchart

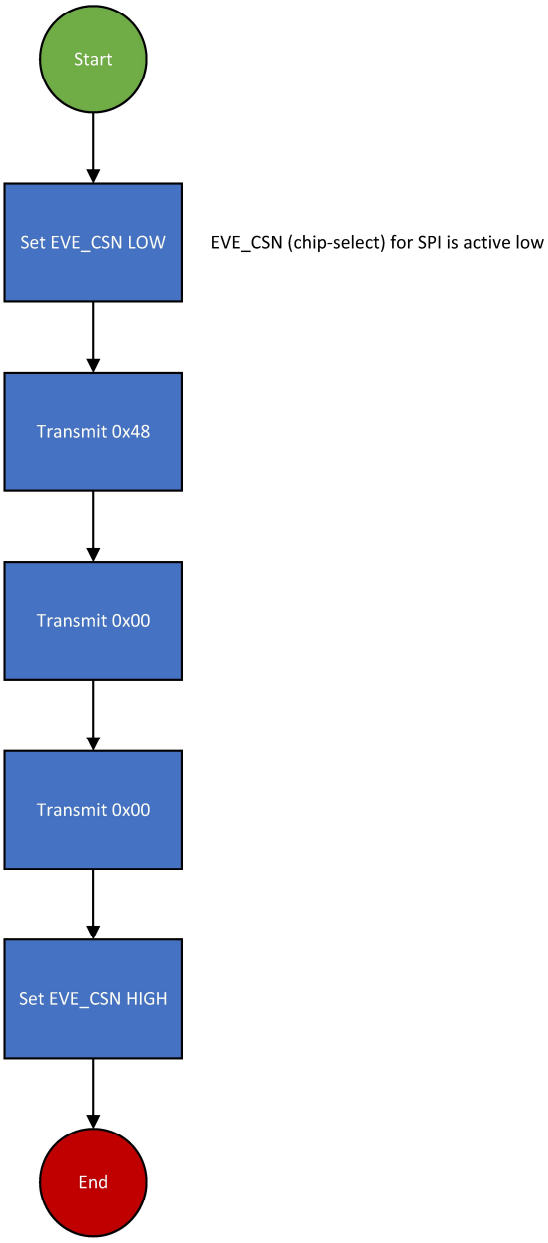


Figure D3: Selecting FT812 Internal Clock Host Command

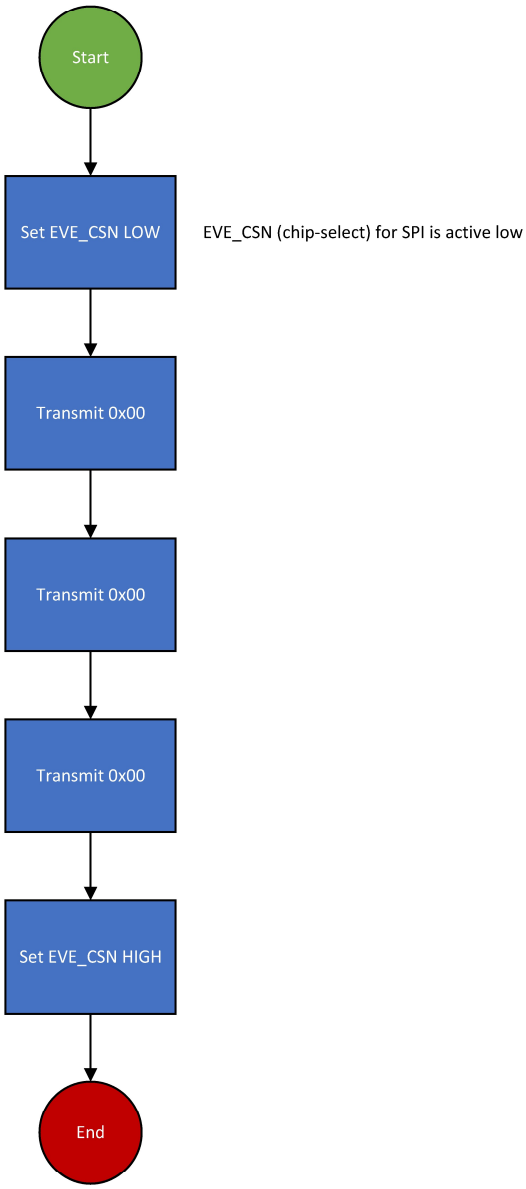


Figure D4: FT812 ACTIVE Command Flowchart

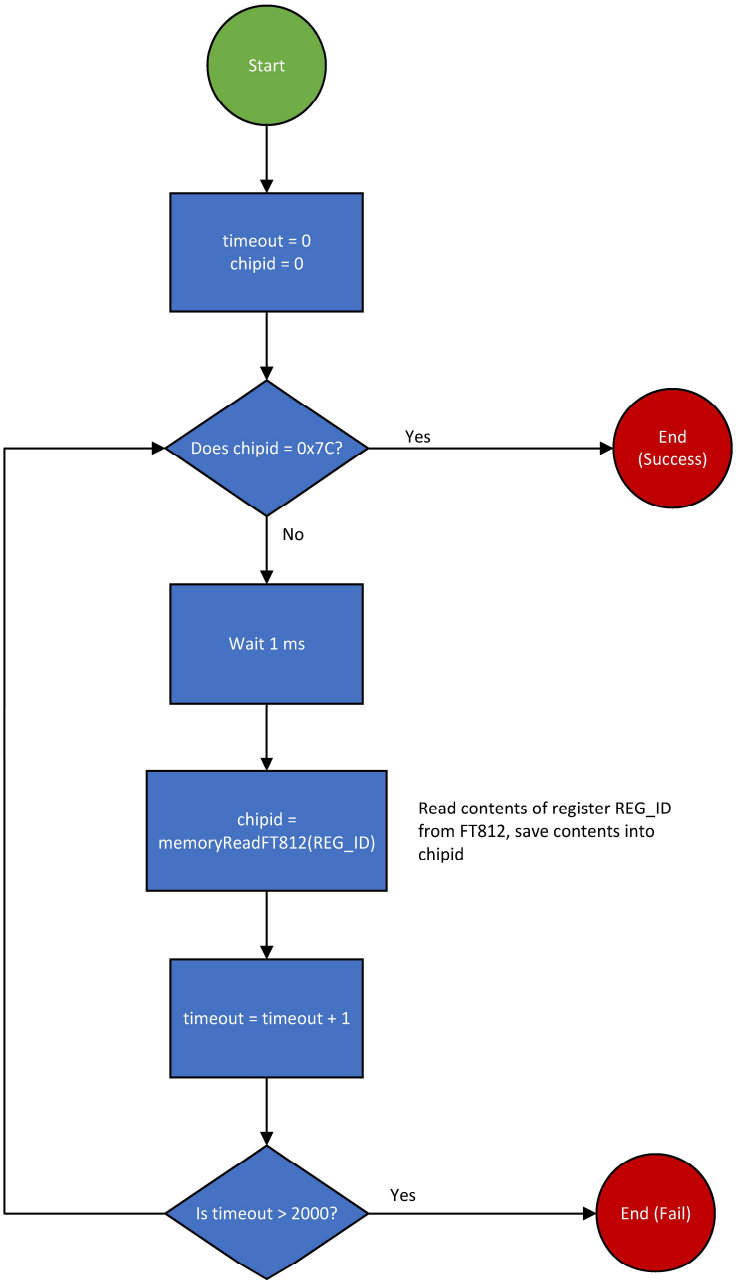


Figure D5: Reading the FT812 Chip ID Flowchart

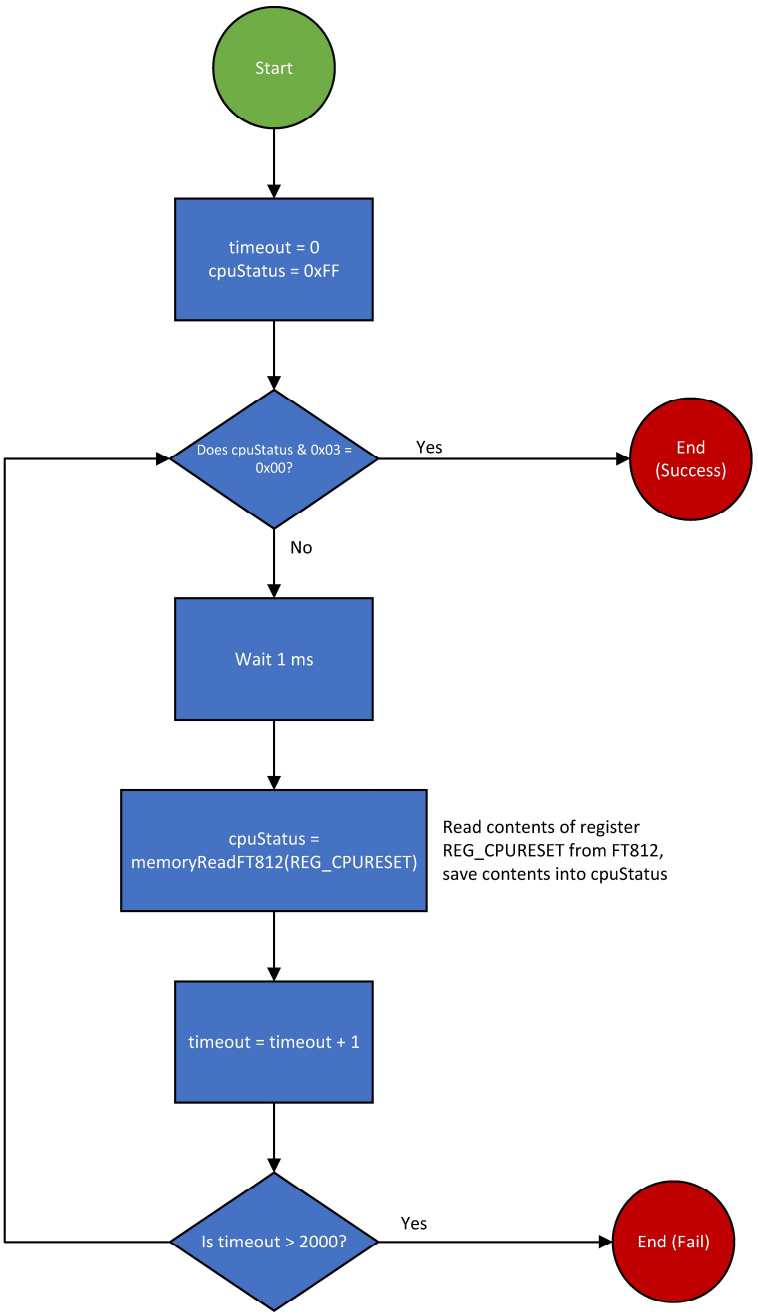


Figure D6: Reading the FT812 CPU Status Flowchart

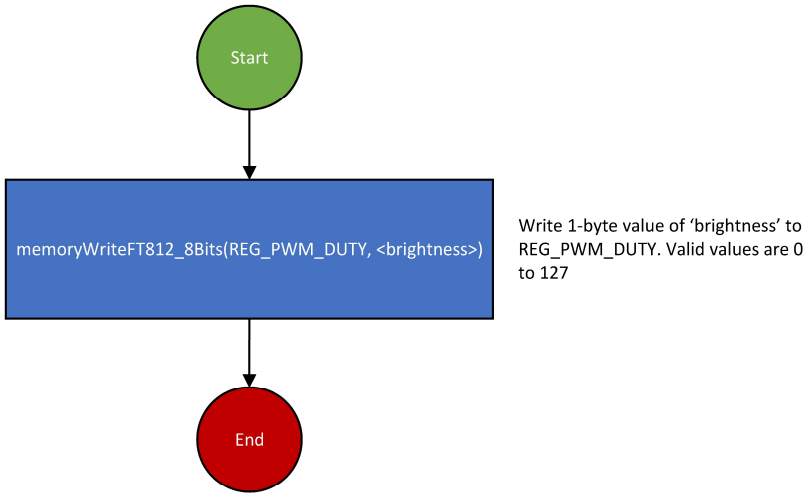


Figure D7: Setting the Display's Backlight Brightness Flowchart

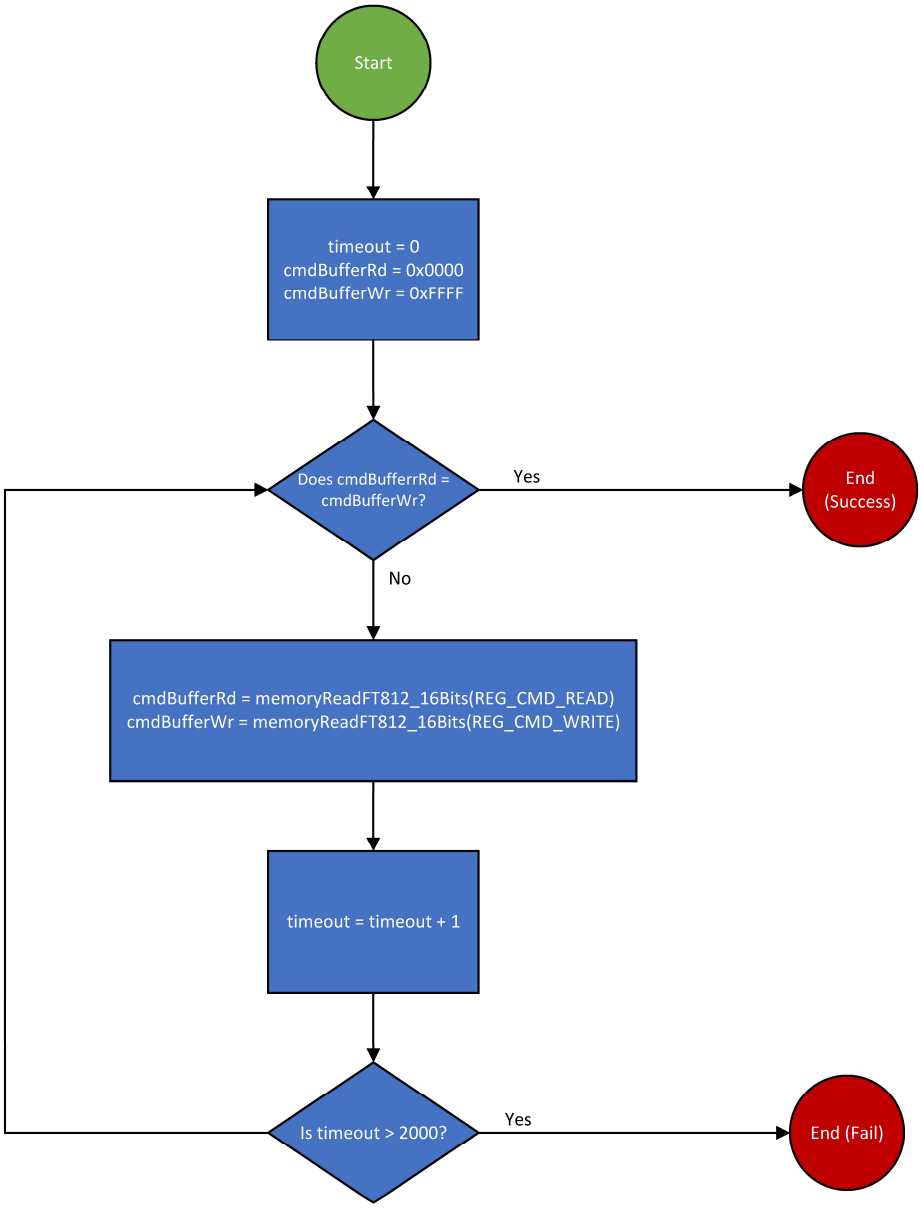


Figure D8: Waiting for a Display Refresh Flowchart

Appendix E

DC_CAN_BASE
ID: 0x500
Interval: 1 second

Variable	Bits	Type	Description
Serial Number	63 .. 32	Uint32	Device serial number, allocated at manufacture (0x00000001)
Tritium ID	31 .. 0	char[4]	"TRIb" stored as a string. msg[4] = 'TRIb'
			msg[0] = 'b', msg[1] = 'I', msg[2] = 'R', msg[3] = 'T'

High word is sent first, within each word, the least significant byte is sent first. Within each byte, the most significant bit is sent first.

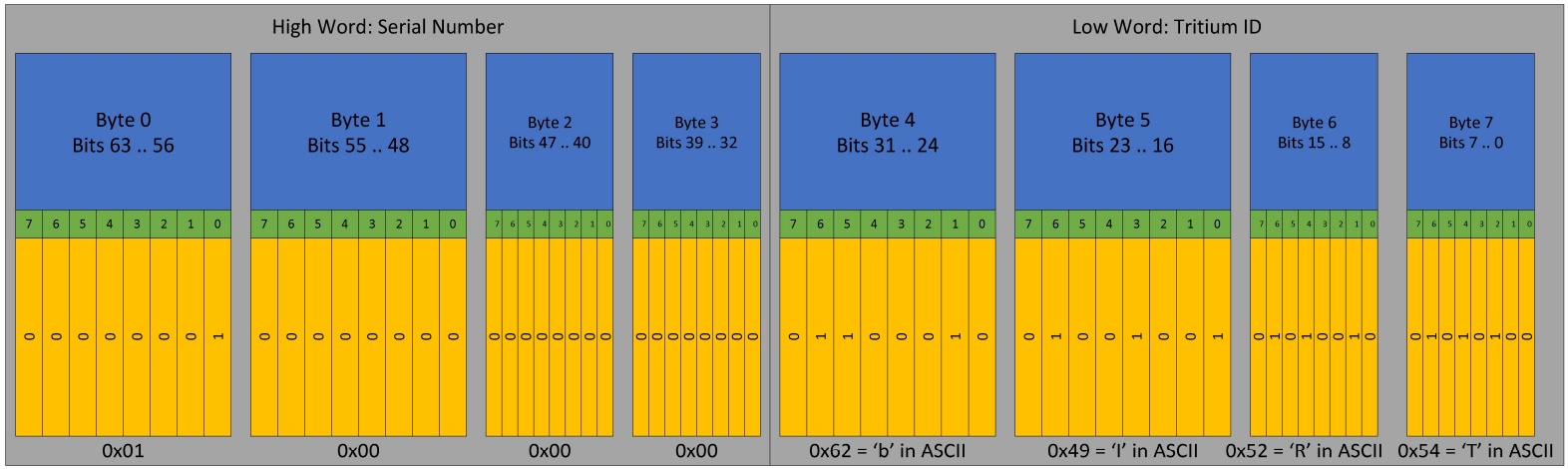


Figure E1: DC_CAN_BASE Message
The above layout shows what the actual message looks like on the CAN bus

DC_DRIVE
ID: 0x501
Interval: 100 ms

Variable	Bits	Units	Description
Motor Velocity	63 .. 32	m/s	Desired motor velocity set point in meters/second
Motor Current	31 .. 0	%	Desired motor current set point as a percentage of maximum current setting

High word is sent first, within each word, the least significant byte is sent first. Within each byte, the most significant bit is sent first.

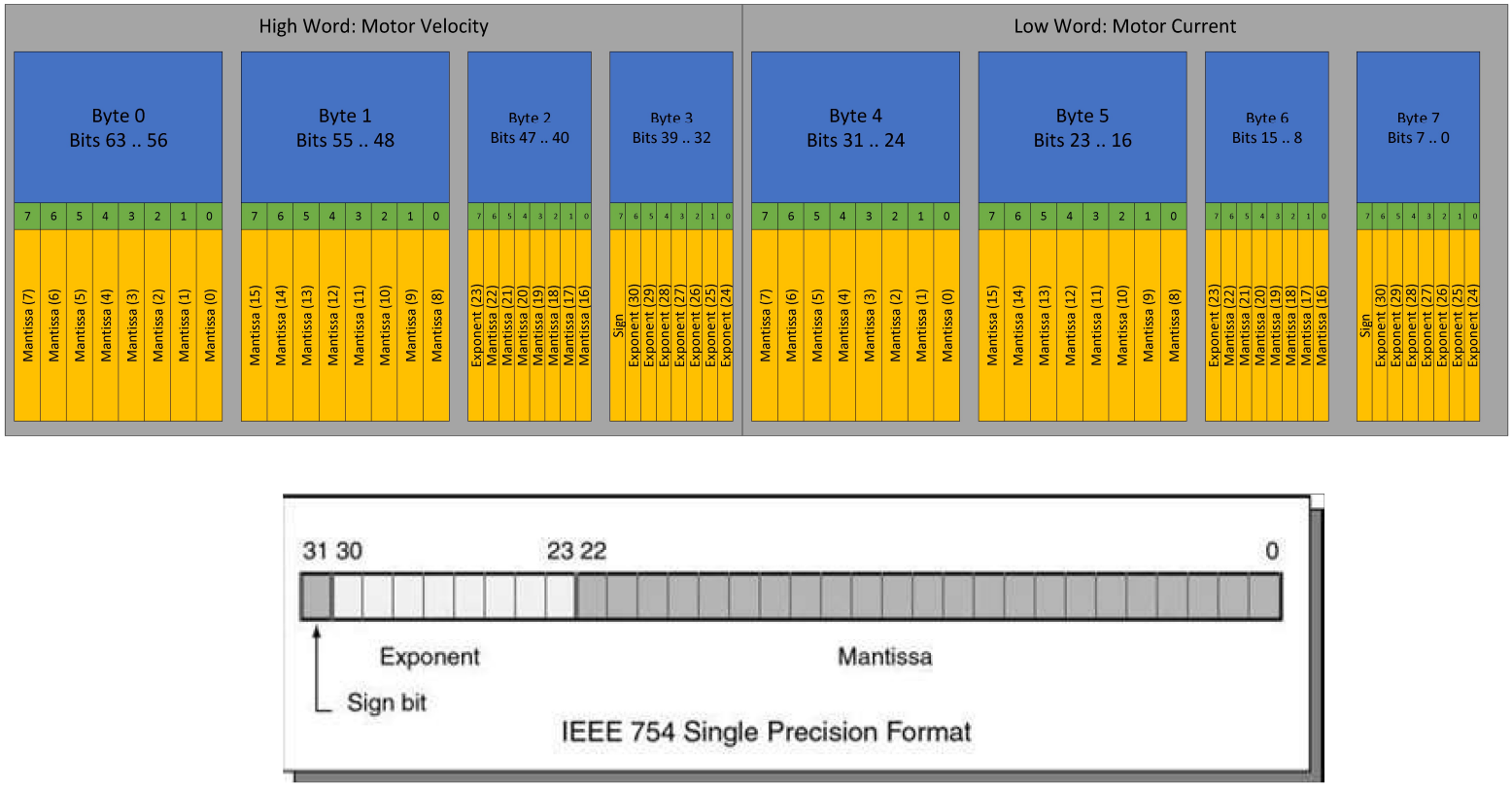
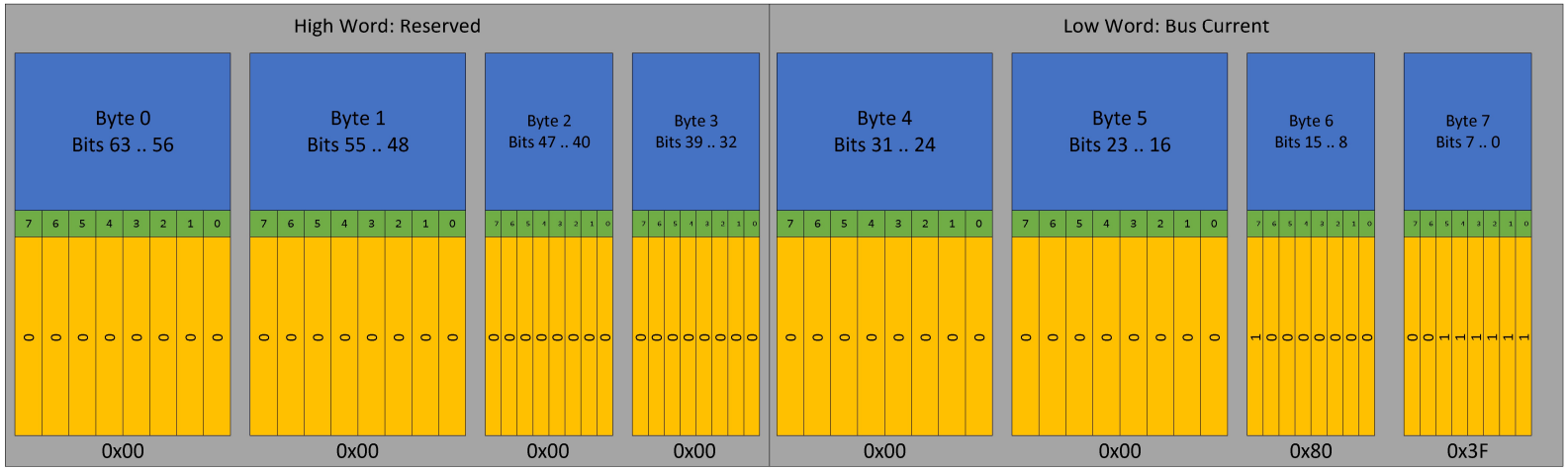


Figure E2: DC_DRIVE Message
The above layout shows what the actual message looks like on the CAN bus

DC_POWER
ID: 0x502
Interval: 100 ms

Variable	Bits	Units	Description
Reserved	63 .. 32	-	-
Bus Current	31 .. 0	%	Desired set point of current drawn from the bus by the controller as a percentage of absolute bus current limit

High word is sent first, within each word, the least significant byte is sent first. Within each byte, the most significant bit is sent first.



Bus current is always 1.0 in IEEE 754, which means 100% to the driver controller and other nodes

Figure E3: DC_POWER Message
The above layout shows what the actual message looks like on the CAN bus

DC_SWITCH
ID: 0x504
Interval: 100 ms

Variable	Bits	Type	Description
Switch Position	63 .. 32	Uint32	Current position of the switch inputs on the driver controls module DB37 connector
Switch Activity	31 .. 0	Uint32	Shows if the switch has changed state since the last time the CAN frame was sent
			1 = Switch has changed
			0 = No change
			Bit positions are identical to the Switch Position bitfield shown above

High word is sent first, within each word, the least significant byte is sent first. Within each byte, the most significant bit is sent first.

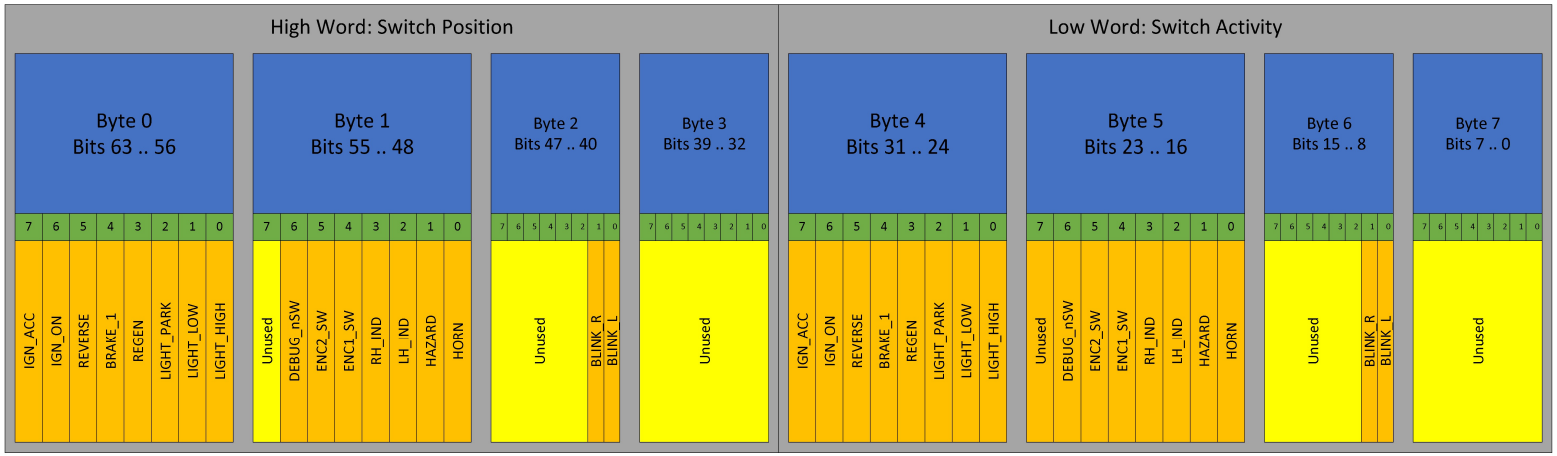


Figure E4: DC_SWITCH Command
The above layout shows what the actual message looks like on the CAN bus

Appendix F

Figure F1: PCB Enclosure Top View

Figure F2: PCB Enclosure Left View of Connectors

Figure F3: PCB Enclosure Front View of Switches

Figure F4: PCB Enclosure Right View of DB37 Connector

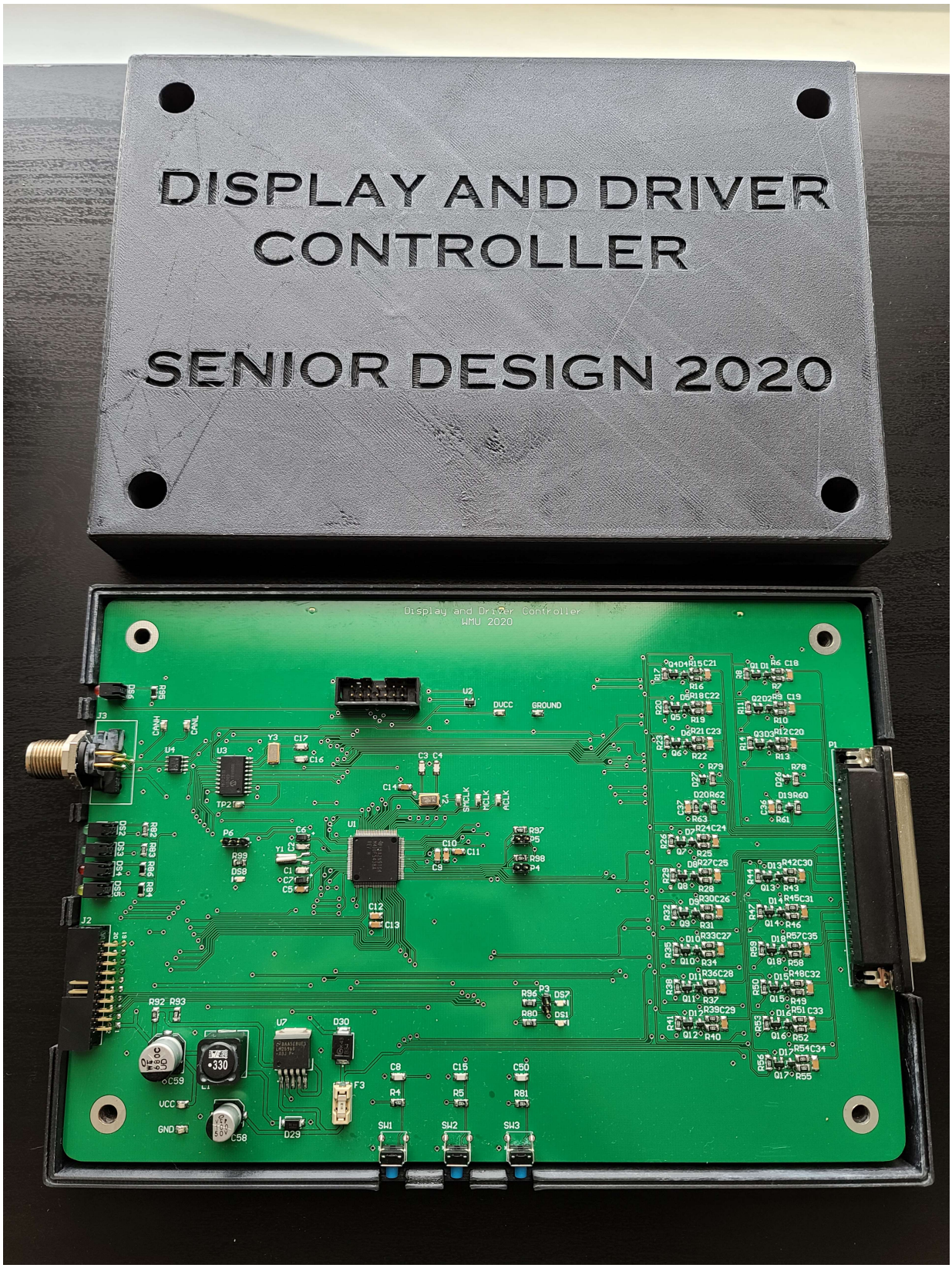


Figure F5: PCB Enclosure with Removed Top Piece

Figure F6: Display Enclosure Top View

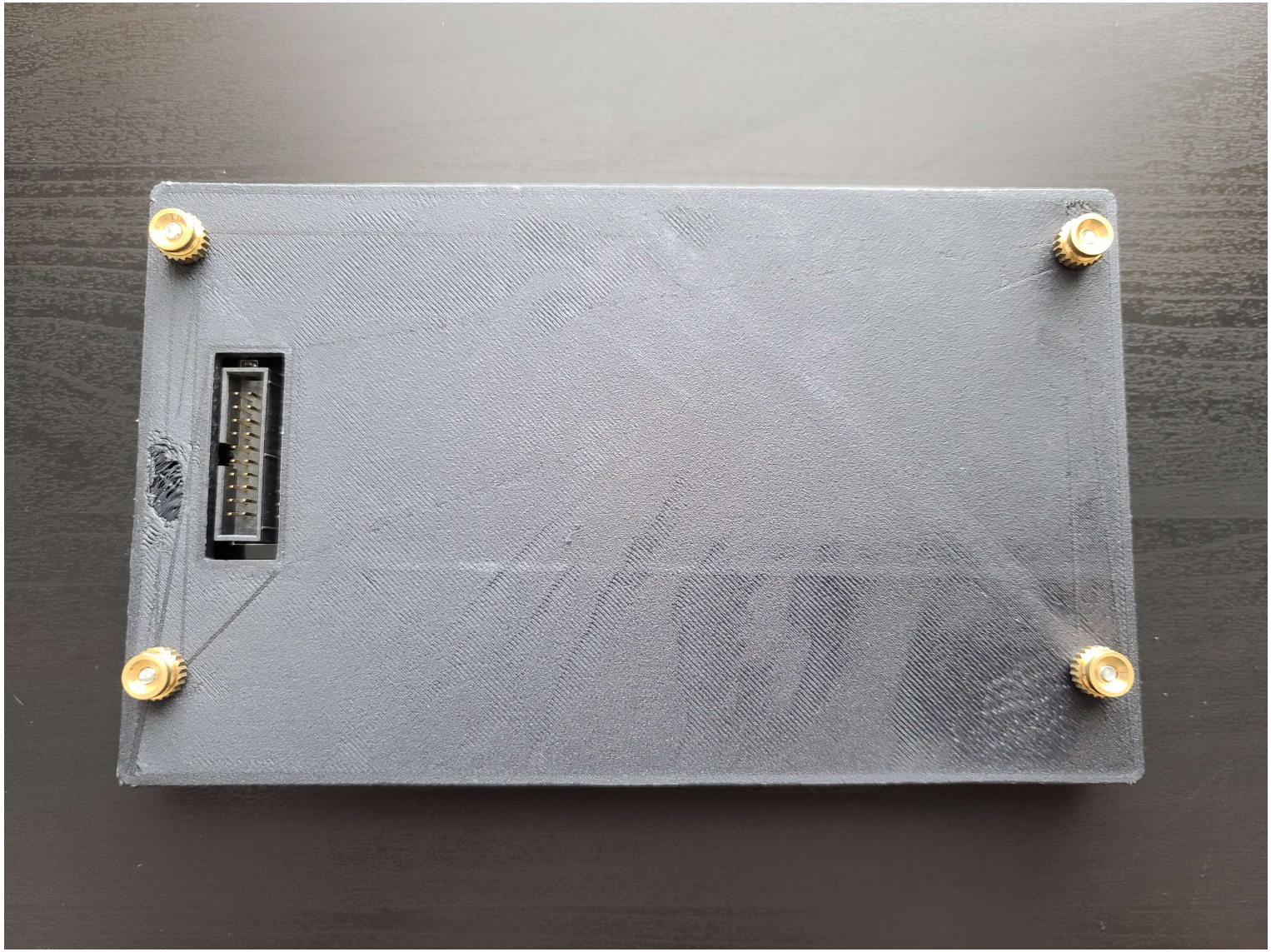


Figure F7: Display Enclosure Rear View of Connector

Figure F8: Fully Assembled and Functional DDC Unit