Generalized **E**lectric **V**ehicle **C**ontrol **U**nit

GEVCU

Version 4.00

LEGAL DISCLAIMER

This manual describes a hardware device produced by EVTV Motor Werks. The Generalized Electric Vehicle Control Unit or GEVCU is an experimental educational device designed to allow individual electric car enthusiasts to explore and learn about vehicle control issues in automotive development.

The device in and of itself performs no particular or specific function and is not designed for commercial or automotive use.

It comes preloaded with one of many versions of the GEVCU open source software project software available at http://github/EVTV/GEVCU. This software, created entirely by enthusiasts outside of the control of EVTV, can be downloaded as source code in its latest versions, modified by anyone anywhere, including the end user or owner of this hardware device, for any reason or at whim, and installed on this hardware and essentially contains the entire functionality of the device.

As such, EVTV Motor Werks has no control over what the device can do, what it is used for, or why. EVTV Motor Werks disclaims any liability arising from the purchase and use of this hardware and makes no claim of fitness for any particular purpose.

The GEVCU hardware is offered solely for the entertainment and educational use of the purchaser. Purchasers of this particular offering of the open source GEVCU hardware agree to defend and hold harmless EVTV Motor Werks from any claims by any party arising from their purchase and use of the GEVCU device.

Table of Contents

1 Introduction

For 40 years and more, individual tinkerers and innovators have been modifying existing automobiles to electric drive and often building electrically powered vehicles from scratch.

The early “controllers” that evolved to replace simple switching and resistive controls to control the speed of the driver motors were mostly pulse width modulated (PWM) devices to provide an averaged DC signal to a series DC motor. These simple chopper voltage control devices were generally referred to as “controllers” and translated driver input from “controls” such as the accelerator pedal, ignition switch, and brake to this motor driving voltage to control its speed and direction – and consequently the vehicle.

There are of course many other types of motors such as separately excited DC motors, brushless DC motors, permanent magnet motors and AC induction motors. Most of these polyphase motors required 3 phase “inverters” to convert the DC power of the battery pack into three phase drive signals varying the voltage (for torque) and frequency (for speed) of the power to the motor to accomplish the same thing.

Most of the DC “controllers” received the inputs directly by wiring from the sensors or controls in the cars.

In recent years, among automobile manufacturers, the use of the Bosche Controller Area Network (CAN) protocol has been adopted for many of the items in the automobile, including the internal combustion engine – often called an **Engine Control Unit** or a **Vehicle Control Unit** which forms the central “brains” of the car. It is connected to various sensors and controls by wire but communicates to other subsystems of the vehicle such as the instrumentation, ABS system, transmission, environmental system using this CAN bus as the common link.

Automakers have eschewed the DC motor in favor of either permanent magnet AC motors or AC induction motors. And the “inverters” developed to drive these were interfaced to this same CANbus model for control input.

Because of the varying nature of the cars, the ECU or VCU would be specifically designed for THAT particular make and model. In this way the Inverters and AC motors can be somewhat generic to work in any car. The intelligence moved OUT of the inverter and into the VCU, which contained all the vehicle specific information.
The VCU would be specifically designed and software specifically written for that vehicle. And all the particulars for that make and model would reside in the software for the VCU. This was hardcoded into flash memory and defined the operation of the vehicle. No end user input or options were provided. Firmware updates or changes are normally accomplished by revising the code, recompiling, and having the controller reflashed with the new binaries at the dealership during normal maintenance.

And so thousands of these VCUs could be flashed when built, with the software specific to that particular vehicle.

The particular VCU would of course be completely inappropriate and non functional in any other vehicle make or model.

As many automakers are experimenting with product introductions of plug-in electric vehicles and hybrid gasoline/electric vehicles, many of the components used in these vehicles are becoming available when the cars are salvaged and are recycled through salvage yards and such online services as eBay.

As such, they will become a resource to individual tinkerers and those converting existing cars to electric drive. Fortunately, most are somewhat generic and almost all of these components were designed to be controlled by CANbus signals from the Vehicle Control Unit that came with the original car.

It would be a serious advantage, to have a more **generalized** vehicle control unit that could produce these CAN commands to drive existing power switching inverters, chargers, dc-dc converters and other equipment gleaned from the many parts available in salvage. But to be truly useful, it should allow some **basic configuration** by the end user allowing these conversions to modify operation of the VCU to accommodate THEIR car without the need to entirely rewrite the software and flash the VCU. Just change a handful of variables specific to the car.

In December 2012, Jack Rickard of Electric Vehicle Television http://evtv.me first proposed a program to develop such a GENERALIZED Electric Vehicle Control Unit

or **GEVCU** using the then just introduced **Arduino Due** platform with an 84 MHz 32-bit ARM CORE3 processor. And he elected to do this as an **open source** project anyone could not only use, but modify further with regards to either hardware or software to meet their own particular needs.

As such, the GEVCU could serve as the central computer or "brains" of any electric car, and flexibly drive ANY available inverters, motors, battery management systems, throttles, brakes, sensors, etc in the car. This modular approach would to some degree commoditize many of the major components of the vehicle, while the specifics were held in a central, open source device that anyone could change and adapt as necessary.

A number of EVTV viewers began contributing code and hardware designs and by late summer 2013, EVTV first drove a 1974 VW Thing, with a Siemens 1PV5135 AC induction motor and DMOC645 inverter from Azure Dynamics, all entirely controlled by the GEVCU. It featured controlled regenerative braking on both throttle and brake, a controllable precharge procedure for applying power to the DMOC, and control of the cooling fans on the liquid cooling system.

Along the way, the original Arduino Due hardware morphed into a somewhat more hardened hardware design capable of surviving the automotive environment, while retaining the full compatibility with the Arduino software development environment.

A selected subset of Arduino input and output pins was brought out to a single weather resistant AMPSEAL 35 pin connector for example. Various strategies and components were used to isolate the inputs and outputs from the multicontroller chip itself to "harden" the device to EMP and EMI and the noise inherent in vehicle 12v systems. But the essential Arduino programming environment and compatibility were retained.

The result was a powerful multicontroller device with TWO programmable CAN bus channels, wireless Internet access, a variety of analog and digital inputs and outputs, and beyond the ability to reprogram the device entirely using the Arduino IDE, the original design allows the end user to easily configure some of the basic aspects of throttle and brake and so forth without really learning to program at all.

This document describes the use and configuration of the **Generalized Electric Vehicle Control Unit**. The multicontroller hardware allows connection of the basic sensor set necessary to drive the car such as throttle signals, brake signals, ignition signal, and control the basic common outputs such as brake lights, fuel level, rpm, power usage, while serving in the central role of converting these inputs to CAN messages for the inverter to actually drive the AC motor and thus the car.

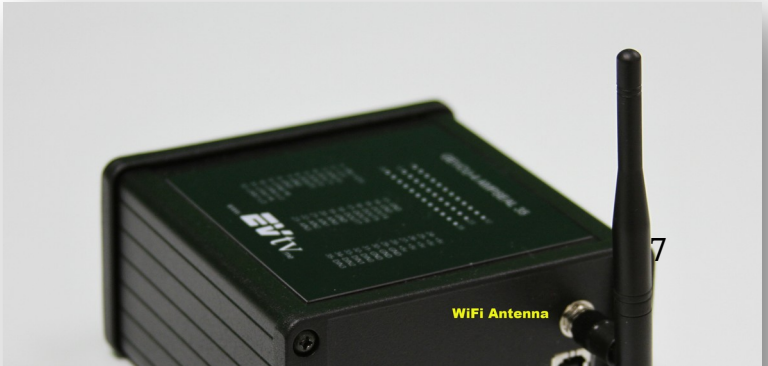
The operation of this device can be modified, within fairly narrow constraints, by a simple configuration “menu” style input that non-programmers can access and make changes to, in order to interface the VCU to the particular car they want to convert.

Note that the GEVCU program is both open software and open hardware with the schematics and board layouts published for all. As such, THIS document ONLY applies to the EVTV produced GEVCU hardware and the software preloaded onto it before shipment.

This document will be updated from time to time to reflect changes in hardware or software that EVTV adopts in their release of the product. But by necessity, it cannot cover changes in hardware or software made by other parties or the end user. This should be obvious.

But we think the end user will find SOME documentation of the baseline EVTV shipped version useful as a starting point.

2 Specifications



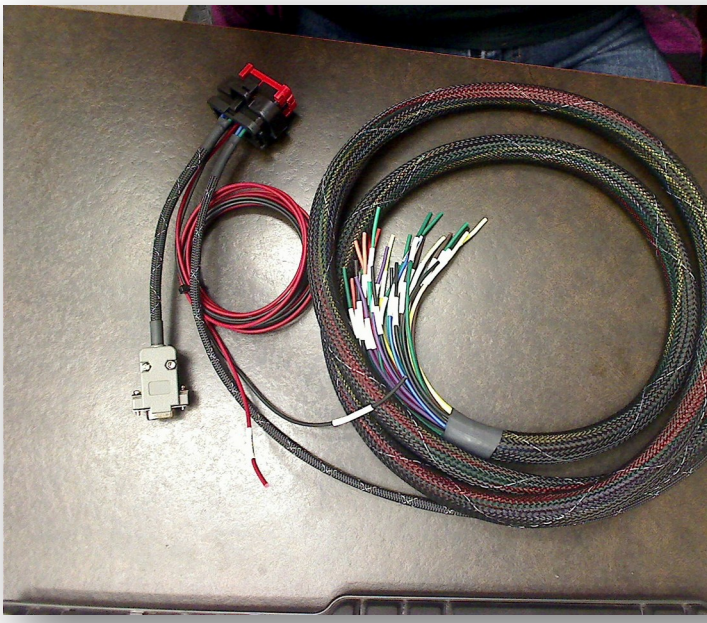


Table 1. AMPSEAL 35 Pin Wire Harness Description

Pin	Color 4.01	Symbol	Description
1	BLK	VIN(+12V)	Positive Power Supply (Ignition KEY)
2	BRN	+12V	Regulated +12V output
3	TAN	DOUT0	Digital Output No. 0
4	VIO	DOUT1	Digital Output No. 1
5	GRY	DOUT2	Digital Output No. 2
6	YEL	DOUT3	Digital Output No. 3
7	RED	GND	Ground
8	ORN	GND	Ground
9	BLU	CAN0 HI	CAN High Channel 0
10	GRN	CAN0 LO	CAN Low Channel 0
11	PNK	CAN1 HI	CAN High Channel 1
12	WHT	CAN1 LO	CAN Low Channel 1
13	BLK/RED	GND	Ground
14	BRN/BLK	GND	Ground
15	BLU/RED	DOUT4	Digital Output No. 4
16	VIO/BLK	DOUT5	Digital Output No. 5
17	GRY/BLK	DOUT6	Digital Output No. 6
18	GRN/RED	DOUT7	Digital Output No. 7
19	RED/BLK	GND	Ground
20	ORN/BLK	AIN0	Analog Input No. 0
21	PNK/BLK	AIN1	Analog Input No. 1
22	TAN/BLK	AIN2	Analog Input No. 2
23	YEL/BLK	AIN3	Analog Input No. 3
24	WHT/BLK	+5V	Regulated +5V output
25	BLK/RED/BLU	+5V	Regulated +5V output
26	BRN/BLK/RED	+5V	Regulated +5V output
27	BLU/BLK/RED	+5V	Regulated +5V output
28	VIO/BLK/RED	+5V	Regulated +5V output
29	GRY/BLK/RED	GND	Ground
30	GRN/BLK/RED	GND	Ground
31	RED/BLK/BLU	GND	Ground
32	ORN/BLK/RED	DIN0	Digital Input No. 0
33	PNK/BLK/RED	DIN1	Digital Input No. 1
34	TAN/BLK/RED	DIN2	Digital Input No. 2
35	YEL/BLK/RED	DIN3	Digital Input No. 3

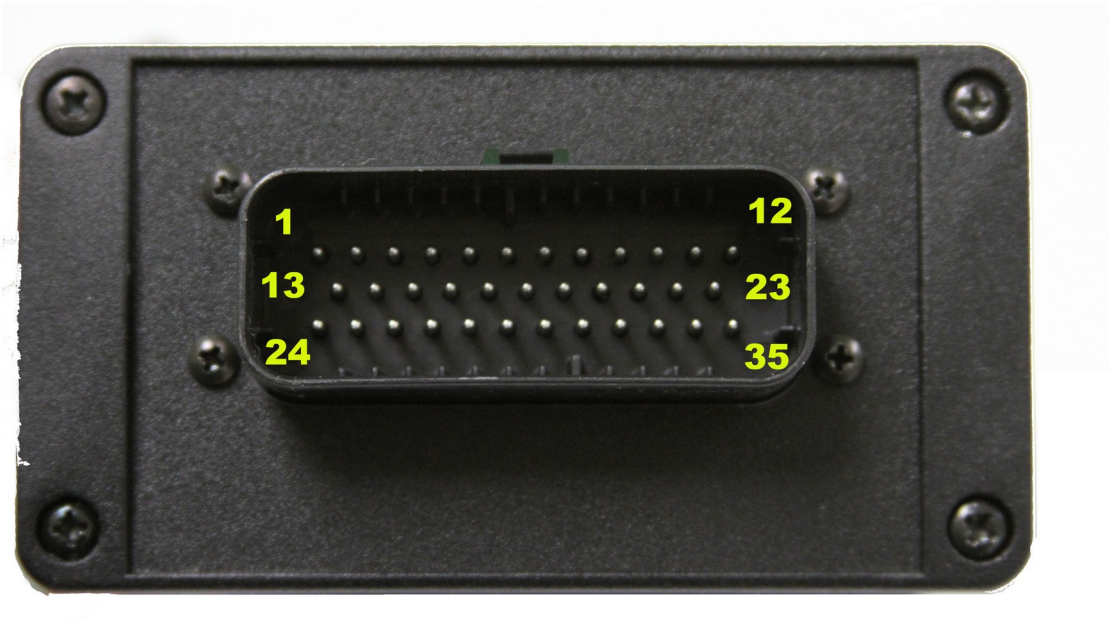


Table 2. Absolute Maximum Rating

Parameter	Symbol	Value	Units
Supply Voltage	VIN(+12V)	15	V
Regulated +12V output	+12V	1.5	A
Regulated +5V output	+5V	1*	A
Digital Outputs	DOUT0... DOUT7**	1.7	A
CAN BUS	CAN0 HI/LO; CAN1 HI/LO	–27 to 40	V
Analog Inputs	AIN0... AIN3	12	V
Digital Inputs	DIN0... DIN3	15	V

*Total value for all pins
**Applying any voltage directly to any of these PINs will cause permanent damage to the GEVCU

MICROCONTROLLER

Atmel SAM3X8E ARM Cortex-M3 CPU
32-bit core
CPU Clock at 84Mhz.
96 KB of SRAM.
512 KB of Flash memory for code
256KB EEPROM for persistent data.
Operating Voltage: 3.3v
Input voltage: 6-16v
CAN network channels: 2
Universal Serial Bus: 1
Isolated Analog Inputs: 4
Isolated Digital Inputs: 4
Isolated Analog or Digital Outputs: 8
Programming Environment: Arduino Due IDE 1.5.4

WIRELESS MODULE

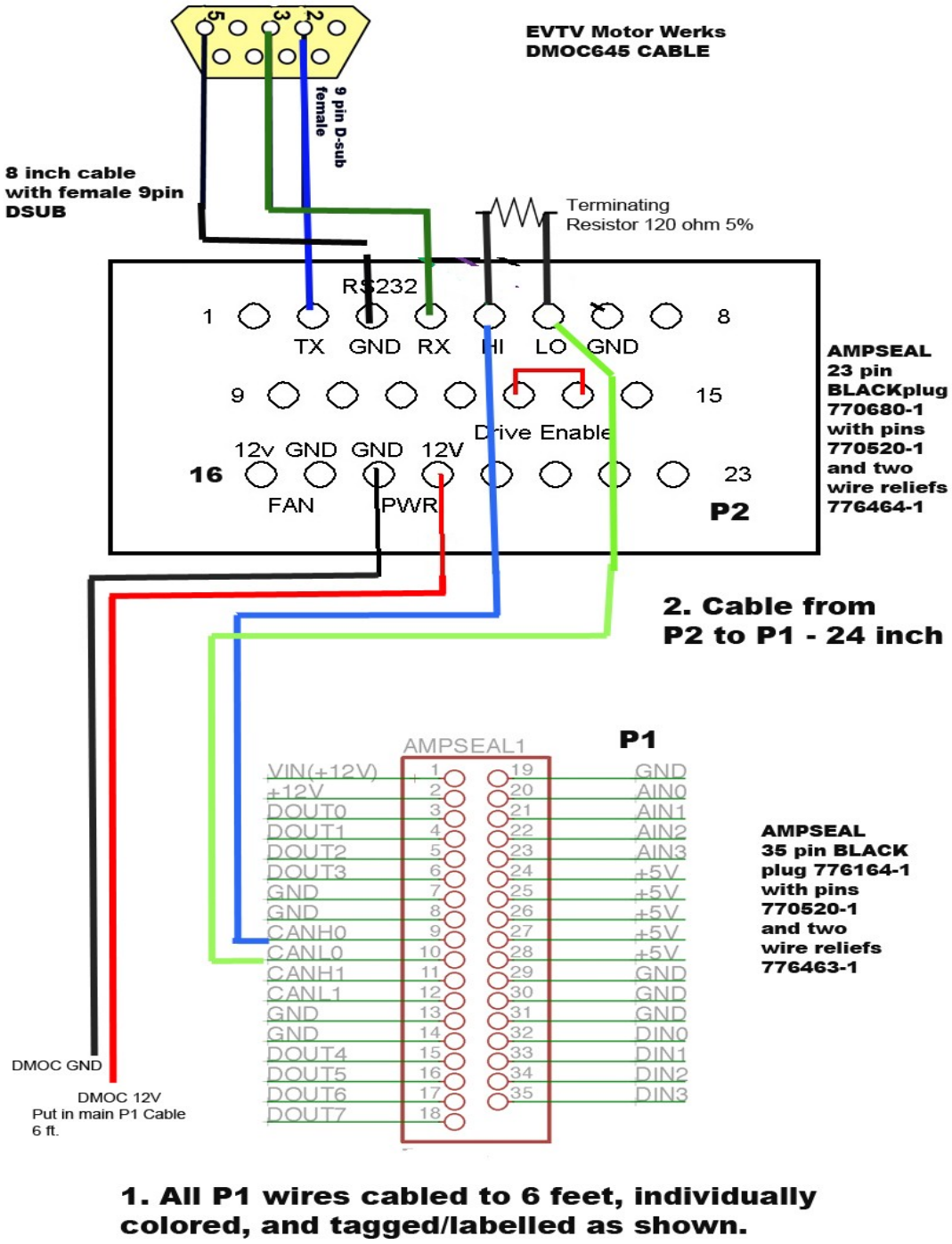
Core CPU: 32-bit RISC ARM7TDMI, low-leakage, 0.13 micron, at 48MHz
Operating Voltage: +3.3V+/-10%
Operating Humidity: 90% maximum (non-condensing)
Operating Temperature Range: -20° to 75°C (-4° to 167°F)
Power Consumption:
Transmit –250mA@16dbm, 235mA@12dbm (typical)
Receive – 190mA (typical)
Power Save mode – 8mA
Power Down mode – 40uA (typical)
RF Connector: SMA reverse polarity
Host Interface: TTL serial interface
Internet Protocols
ARP, ICMP, IP, UDP, TCP, DHCP, DNS, NTP, SMTP, POP3, MIME, HTTP, FTP and TELNET
Security protocols
SSL3/TLS1, HTTPS, FTPS, RSA, AES-128/256, 3DES, RC-4, SHA-1, MD-5, WEP , WPA and WPA2
Protocols accelerated in hardware: AES, 3DES and SHA

Wireless Specifications:

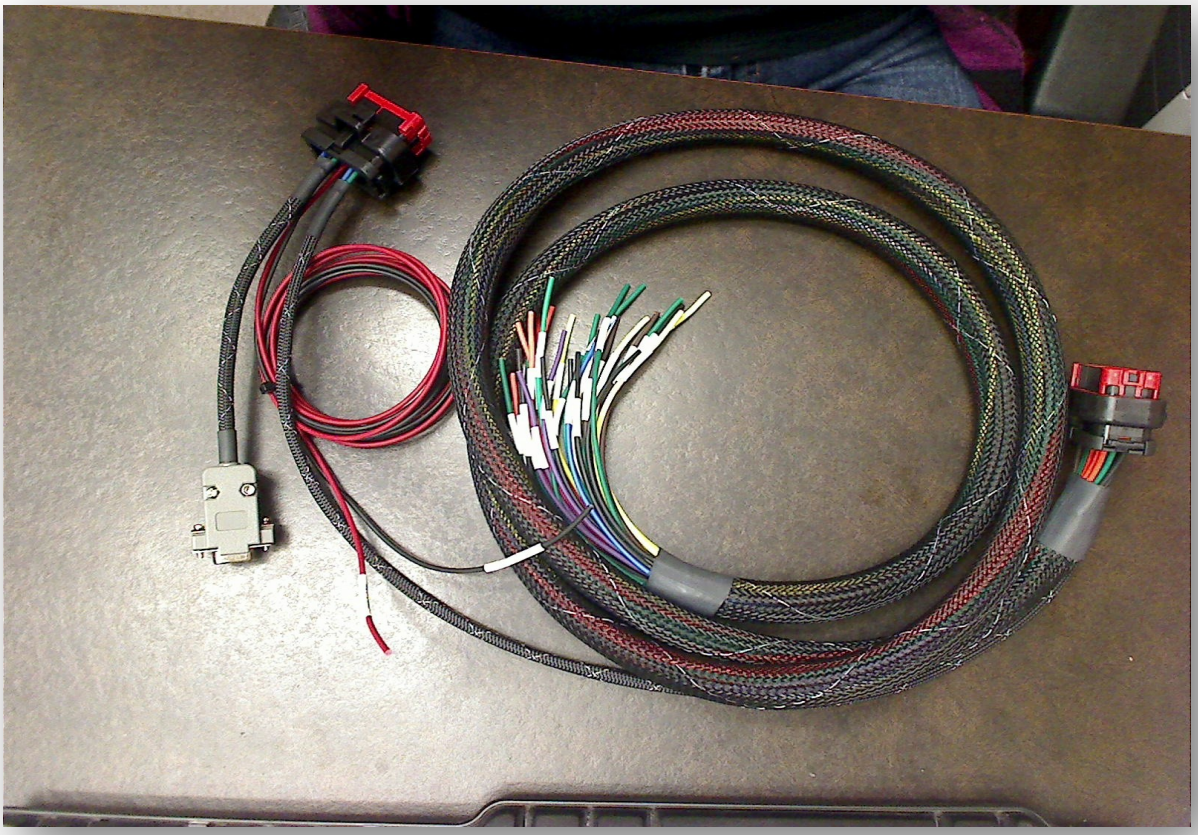
Standards supported: IEEE 802.11b/g
Frequency: 2.412-2.462GHz
Channels: 11 channels

Host Data Rate: up to 3Mbps in serial modeSerial Data Format (AT+i mode):
Asynchronous character; binary; 8 data bits; no parity; 1 stop bit

3 Wiring and Connections



The GEVCU must of course be connected to the car and the inverter/controller by wire. The entire interface is accomplished through a weatherized AMPSEAL 35-pin connector which is quite conventionally used in automotive applications.



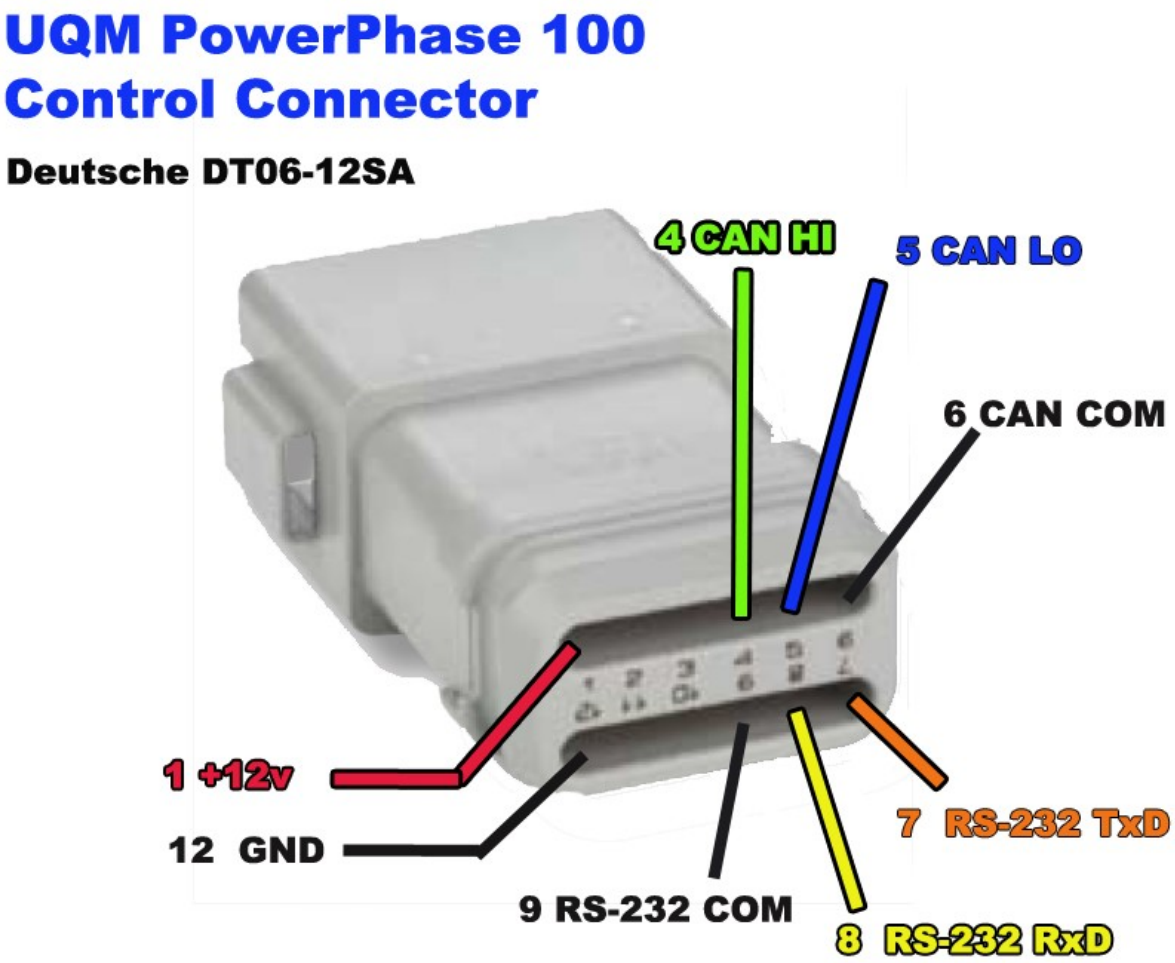
Provided with each GEVCU unit from EVTV is a basic wiring harness. The original genesis of GEVCU was the Arduino Due which featured a very large number of input and output pins.

The AMPSEAL 35 provides but a subset of that but features inputs for 12v vehicle power, two CAN lines to the DMOC645 AMPSEAL 23-pin connector, four analog inputs, four digital inputs, up to eight digital outputs and isolated 12v and 5v outputs to power sensors.

This harness is a bit specialized for the Azure Dynamics DMOC645 Digital Motor Controller and so features an AMPSEAL 23-pin connector for that unit. This connector provides the two CAN bus lines, an enable jumper, terminating resistor for the CAN bus, 12v vehicle power for the DMOC645, and breaks out an RS-232 serial connector to the DMOC645. This can be useful in the event the firmware

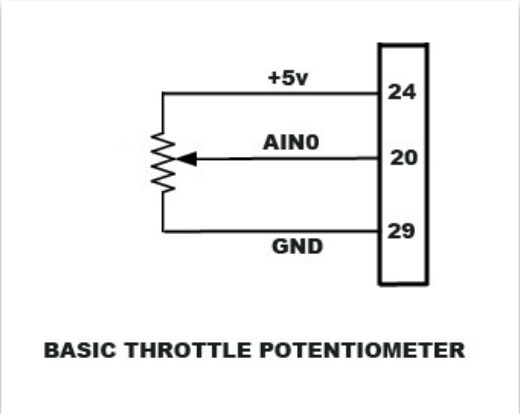
should need to be reflashed or internal variables modified with a program titled ccShell. ccShell and instructions for connecting to the DMOC645 for this purpose can be found at http://forums.evtv.me.

This cable would need to be modified for other controller/inverters such as the UQM Powerphase 100.



While the GEVCU has a limited number of digital and analog inputs and outputs, they are really quite capable of handling a number of necessary duties in controlling a vehicle. And the unit does feature TWO CANbus ports, one rather dedicated to the DMOC645 but the other quite free to interact with the vehicle. In this way, GEVCU can be extended with other CAN equipped multicontrollers used for battery monitoring, informational displays, charging issues, etc.

BASIC THROTTLE WIRING



BASIC THROTTLE POTENTIOMETER

As an example, we will describe some basic throttle wiring for the GEVCU as this is the minimum necessary application to control a motor.

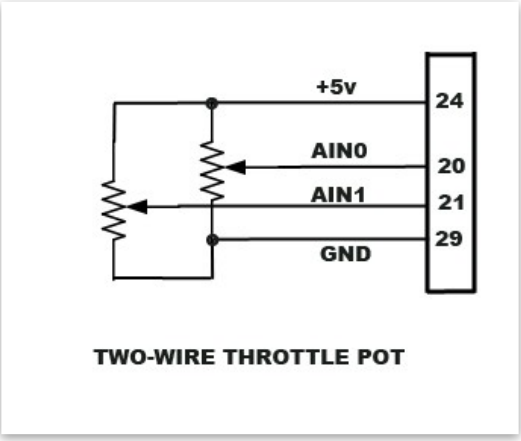
A basic throttle potentiometer is shown in the diagram . The potentiometer is a variable resistor that basically "divides" a voltage based on varying the point along a linear resistance where the voltage is sampled. This signal output is tied to our 1st analog input line AIN0 at pin 20 of the AMPSEAL 35. The voltage to be divided is provided by tying one end of the pot to our +5v output available on several pins for convenience – in this case pin 24. The other end of the pot is tied to our reference ground at pin 29.

In this way, as you press on the accelerator, the potentiometer is varied, increasing the voltage from zero to five volts at maximum pedal. In practice, this is usually something like 0.80v to 4.50 v.

Hall Effect pedals work on a different principle than variable resistors – magnetic inductance. But from a wiring standpoint are no different. They still need a 5v power source and reference ground. And they still provide a signal output that we tie to one of our analog inputs.

Many modern throttles have two pots or hall effect modules and provide TWO signal outputs.

This is to provide a "sanity check" to make sure the throttle input is valid. For example, if the signal output and the 5v lines were shorted, this might be read as a max throttle input and the vehicle accelerate uncontrollably.



TWO-WIRE THROTTLE POT

Two wire throttles provide two different outputs and they usually are NOT identical. For example, one output might vary from 0.8 to 4.5v while the second

varies from 1.5v to 3.4v. The sanity check association in comparing those two signals must be handled in software. Indeed, one signal might vary from 0.8 at idle and 4.5v at max, while the other is inverted, showing 3.5v at idle and 1.2v at max pedal. Again, this association must be controlled by the software. GEVCU software currently supports all of these modes. You will have to select 1 wire or 2, and whether the values are linear or inverted.

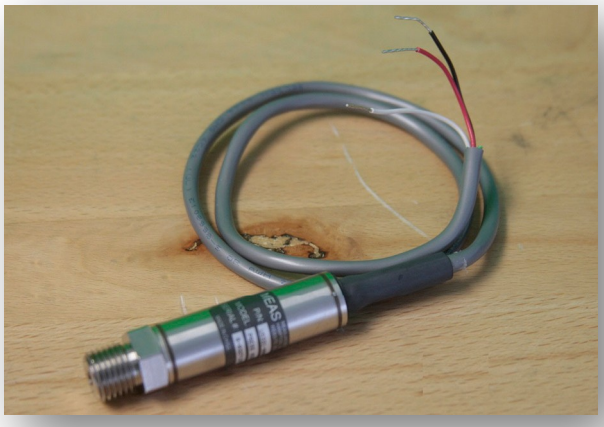
USING EXISTING VEHICLE DRIVE BY WIRE THROTTLES

Many modern vehicles already feature an accelerator that is wired for producing these signals. It is relatively easy to locate the signal lines and tap into them for connection to the GEVCU. Generally it is good practice to use both signal outputs where available. That the pedal gets 5v from the normal ECU is not a problem as 5v is more or less 5v regardless of source. But it does have to be referenced to the same ground. So generally, you want to tap both accelerator outputs AND connect the associated return from the pedal to one of the GND input pins on the GEVCU AMPSEAL 35.

BRAKES

Most electric cars map both acceleration and regenerative braking to the throttle, using the first part of pedal travel for regenerative braking and the latter portion of pedal travel for forward acceleration. You will rather quickly learn to not only drive this way, but usually decelerate to a stop using the regenerative braking and it becomes kind of a single-footed driving pattern that most drivers find very controllable and natural.

But many want regenerative braking to assist slowing the car when using the brake and some do NOT like the use of regenerative braking on the throttle at all and ONLY want it applied during actual braking.

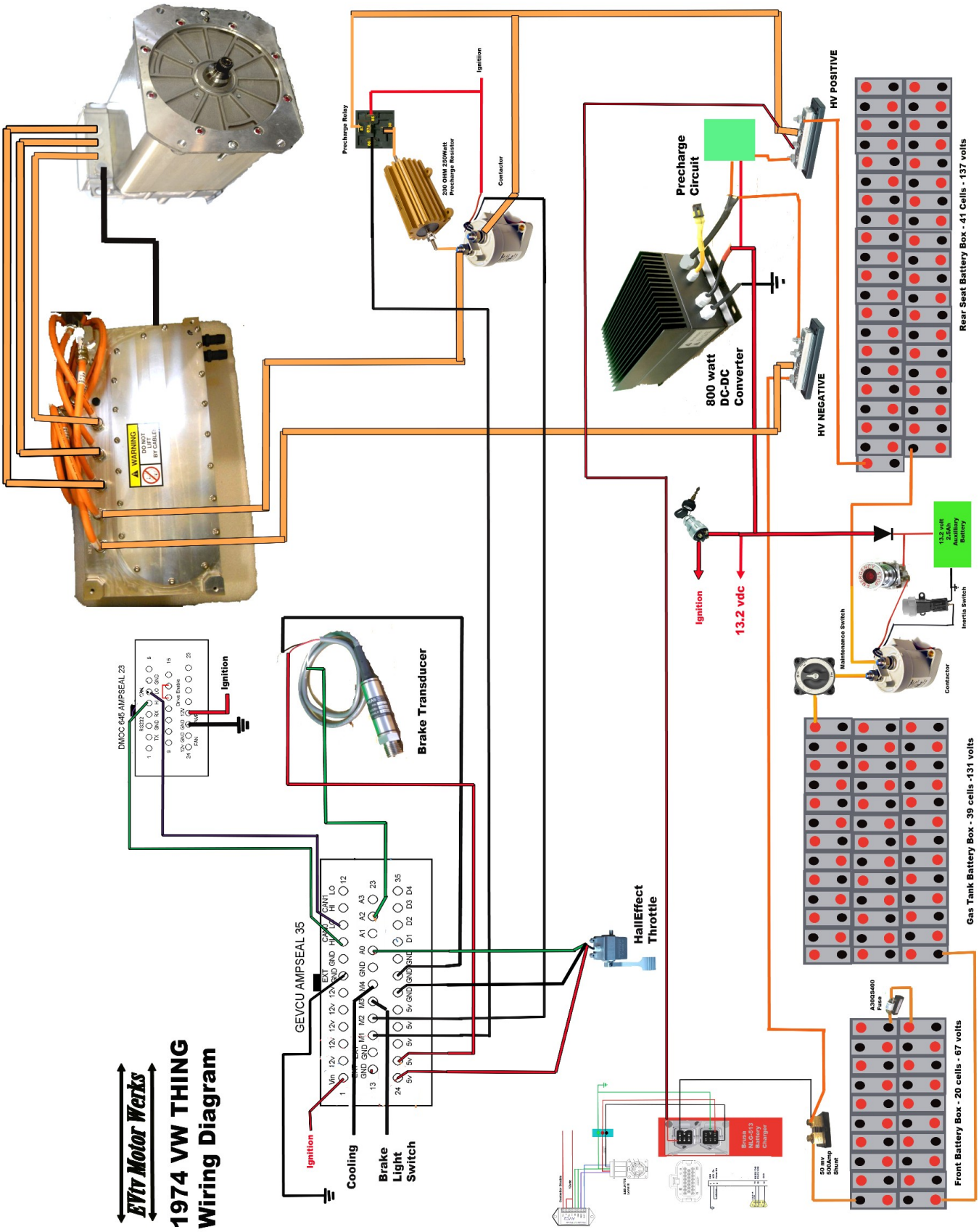


The only way we have found to controllably vary the DEGREE of regenerative braking from the brake pedal is to use a hydraulic pressure transducer that operates

essentially just like a throttle pot. These transducers again require a 5v supply and ground return, and again provide a signal output from 0.5 to 4.5v typically, increasing as the pressure in the brake lines is increased. The transducer has to be connected to the existing hydraulic brake lines.

By convention, GEVCU uses AIN0 for 1 wire throttles, AIN0 and AIN1 for two wire throttles, and AIN2 for brake inputs.

The following diagram shows a more complete wiring view of the GEVCU as installed on the EVTV VW THING build.


EVtv Motor Werks
1974 VW THING
Wiring Diagram
DMOC 645 AMPSEAL 23
GEVCU AMPSEAL 35
Ignition
Cooling
Brake Light Switch
Brake Transducer
HallEffect Throttle
Precharge Relay
200 OHM 250Watt Precharge Resistor
Contactor
Precharge Circuit
800 watt DC-DC Converter
HV POSITIVE
HV NEGATIVE
Rear Seat Battery Box - 41 Cells - 137 volts
Gas Tank Battery Box - 39 cells -131 volts
Front Battery Box - 20 cells - 67 volts
13.2 vdc
13.2 volt 2.5Ah Auxiliary Battery
Inertia Switch
Maintenance Switch
A3KQS400 Fuse
50 mv 500Amp Shunt

4 USB SERIAL PORT INTERFACE

The key concept in the GENERALIZED vehicle control unit is that it is **generalized**. That is, it can be configured and used in a variety of different vehicles using different drive train and vehicle components.

The open source nature of the GEVCU software allows anyone with basic C++ coding skills to extend this ad infinitum. However, for most users, C++ is a bridge too far. You can easily use GEVCU by changing a few simple variables requiring no programming knowledge whatsoever.

The design philosophy is to avoid specialized software programs that are operating system dependent and require updating. To accommodate future use of GEVCU, we cannot predict what operating systems will be used and we do not want the overhead of updating a specialized "configuration program" in any event. For far too much of our EV equipment, we find outdated buggy software running on obsolete and in some cases hardly available operating systems in order to change a few simple variables.

But everyone that touches GEVCU wants something more and having it be powerful and extensible is certainly desirable. We see three basic interfaces for non-programmer users.

1. USB Serial Port terminal program
2. Any web browser via 802.11b/g WiFi wireless.
3. Mobile tablet or phone interface via CANbus/OBDII connector.

In this way, we can avoid depending on specific operating systems or programs beyond a basic serial terminal program or web browser running on ANY device. And at the same time provide for gorgeous graphic interfaces to actually serve as a Tesla style interface for our vehicles.

The most basic albeit primitive of these three is the USB serial port terminal program. GEVCU features a printer style USB port on the rear panel simply because these appear to be the most physically durable and robust connectors for USB.

On powerup, GEVCU will interface via USB serial port using simple ASCII characters and line feeds. You can interact usefully with GEVCU via any serial terminal program on any laptop or other computer device. This USB bootstrap operation is a central tenet of the Arduino concept.

Serial communications dates back to the early modems and electronic bulletin boards and has its own quirks and foibles. It is so dated that neither Microsoft nor Apple actually include a serial terminal program with their operating system. But because the need for basic serial communications never quite goes away, terminal programs for both are still readily and in most cases freely available.

There are some basic terminal settings that must be set on most terminal programs in order to “talk” to the GEVCU.

Data Rate: 115,200 bps
Data bits: 8
Parity bits: None
Stop bits: 1
Character set: ASCII.

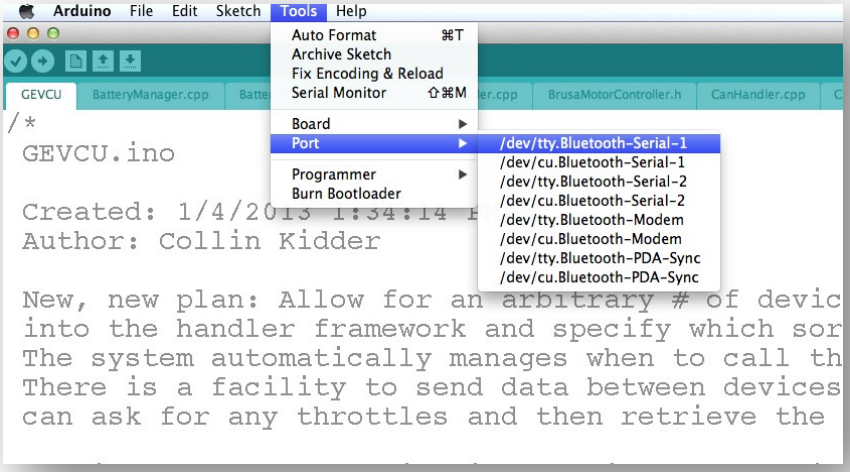
And so to configure GEVCU at its most basic level, you must first configure an ASCII serial data terminal program. Many of these offer many features allowing you to set screen color, font , text size, color. Etc. They can also allow you to “capture” text sent over the port.

GEVCU was born of the Arduino Due educational platform and many actually develop C++ code via the Arduino Integrated Design Environment or IDE. This IDE actually includes a terminal program.

If all else fails, Arduino is freely available for download and installation on Windows, Mac OS X, or Linux fully featured and entirely free of charge. So it may be the easiest way to get and install a terminal program for many users.

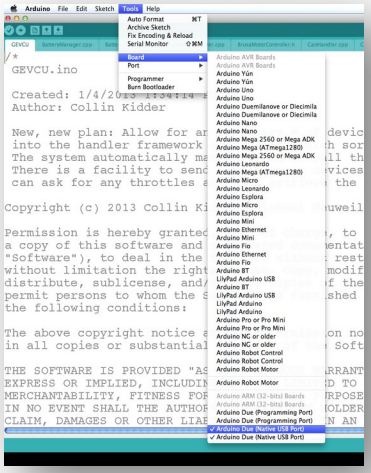
GEVCU is currently usable with Arduino Version 1.5.6.r2.

After downloading and installing Arduino. At the top of screen you will find menus. Select TOOLS, and in the submenu PORT. Then select the hardware USB port you have connected to GEVCU.



For Mac OSX or Linux, the available USB ports will appear as CU or TTY entries.

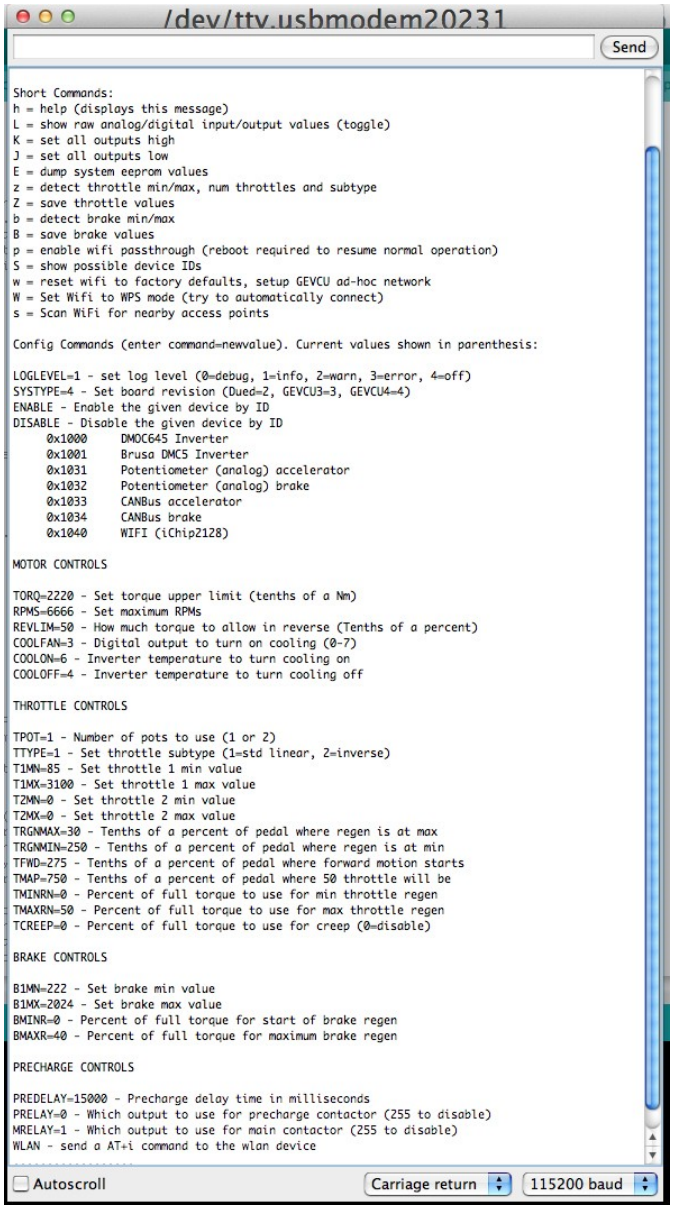
For Windows, more likely something like COM1 or COM2.



Once the port is selected, use the same TOOLS menu to select BOARD for the **ARDUINO DUE NATIVE USB** port. GEVCU actually does not have a PROGRAMMING port but uses the native USB port

Once the port and board are selected, you can use the **SERIAL MONITOR** entry on the **TOOLS** menu to bring up a separate serial terminal window.

This window will feature a data entry field at the top of the screen and a larger display area for text received from the GEVCU.



The only thing you will see initially is that most likely a period
Every few seconds, you will see another one appears on screen.
heartbeat. This is the GEVCU

If you enter a question mark (or
lower case h) on the data entry
field and press enter or return
key you should see a full
GEVCU menu in all its gory
and hideous glory.

If you actually get garbage or
garbled text, make sure the
data rate in the lower right hand
corner of the terminal program
is set for 115,200.

This screen allows you to configure
and use essentially all the
features of the GEVCU by entering
commands in the data field area.

Note that not all menu features
will appear until the associated
module is enabled as described in Section 5.

The Browser/Wireless configuration interface and the CANbus OBDII configuration interface will be further described later in this manual.

5 MODULE SELECTION AND INITIALIZATION

With power and flexibility, unfortunately comes complexity. GEVCU is initially designed to drive the Azure Dynamics Force Drive system with a DMOC645 controller and a single input throttle. The very basics were to take throttle commands and convert them to CANbus signals to drive the DMOC645 to drive the motor. Simple enough.

But the immediate vision of GEVCU from the start was to serve as a modular, object oriented software program that could drive a VARIETY of inverter/controllers and motors. Rather immediately someone wanted such a device for a BRUSA controller. And someone else wanted to interface with a THINK vehicle battery management system. And use it to drive a TCCH charger via CAN bus. And so on and on.

There are also a variety of already drive by wire throttles in a variety of modern vehicles out there. Most have TWO inputs that vary similarly, but usually not identically. And some throttles actually have a CAN output or are converted to CAN signals by the ECU in the original vehicle.

As a result, GEVCU could conceivably grow into dozens of object modules to support various throttles, brakes, battery management systems, and most of all a variety of inverters/controllers.

Obviously, while everybody needs SOME of these object modules no one will ever need ALL of them on the same vehicle. Some form of object module management is needed.

The current system is admittedly very awkward. Until we get it fixed, you can turn on a module via the serial port with an ENABLE command and you can conversely turn it off with a DISABLE command. This does not actually take effect until you power cycle the GEVCU. That is, completely remove power from the unit and then bring it back up with 12v power.

This is best done by disconnecting the AMPSEAL 35 connector entirely and using power over the USB connector. Simply enable the desired module, then unplug the USB connector. Then plug it back in bringing up the system. The new module will be mapped to the system EEPROM and this module will load automatically in the future until it is removed with a DISABLE command.

There are also several versions of the GEVCU hardware out there already and more likely to come. So we need to set the SYSTYPE and power cycle that also as the first order of business.

With a serial terminal program connected to the USB port on the GEVCU.

LOGLEVEL=1 Sets the onscreen reporting level of messages

SYSTYPE=4 Current EVTV GEVCU hardware

System type updated. Power cycle to apply

Power reset

ENABLE=0x1000 Enable the DMOC645 inverter

Successfully enabled device.(%X, %d) Power cycle to activate.

Power reset

ENABLE=0x1031 Enable a normal potentiometer/hall effect style throttle

Successfully enabled device.(%X, %d) Power cycle to activate.

Power reset

ENABLE=0x1032 Enable a normal potentiometer or hall effect brake input

Successfully enabled device.(%X, %d) Power cycle to activate.

Power reset

ENABLE=0x1040 Enable the wireless web server configuration module

Successfully enabled device.(%X, %d) Power cycle to activate.

Power reset

The modules currently available:

DMOC645	0x1000
BRUSA_DMC5	0x1001
CODAUQM	0x1002
BRUSACHARGE	0x1010
TCCHCHARGE	0x1020
POTACCELPEDAL	0x1031
POTBRAKEPEDAL	0x1032
CANACCELPEDAL	0x1033
CANBRAKEPEDAL	0x1034
ICHIP2128	0x1040
THINKBMS	0x2000

Available modules can be found in the current source code file **DeviceTypes.h**

6 PRECHARGE CONSIDERATIONS

Almost all power switching devices feature a set of input capacitors to buffer the supply voltage to the power switching electronics. This ensures a stable input voltage for switching purposes, at least at the frequencies common in those circuits.

The nature of capacitors is that they resist any change in voltage by providing or absorbing current. And this leads to a bit of a problem that comes up in electric vehicles in a variety of places – excessive inrush current.

When we first connect a battery pack to any inverter or PWM controller, the voltage of the capacitors will be zero while the voltage of the pack might be as high as 400v. The capacitor will absorb current in an attempt to maintain zero volts until it is forced to 400v. And so the capacitor is said to CHARGE. If the voltage is then removed, the capacitor will PROVIDE current attempting to maintain the voltage until it is discharged.

The amount of current is a function of the applied voltage and the SIZE in FARADS of the capacitor with any resistance serving to limit current into the capacitor. Because these tend to be large capacitors, they can absorb a large amount of initial current for a brief time before their voltage is equalized to the applied pack voltage.

This can lead to very brief, but often HUGE inrush currents when the pack voltage is applied. These currents can be SO large that they arc weld the contacts on contactor relays, and indeed too often result in the destruction and failure of the capacitors themselves – potentially destroying your DMOC645 inverter.

The solution to this is precharging the capacitors up to the pack voltage through some sort of current limiting device – typically a resistor. By ohms law, a resistor will allow a certain level of current based on its resistance and the applied voltage. I = E/R where I is the current, E is the voltage, and R is the resistance.

So for example, if we have a 400v pack, and we precharge through a 100 ohm resistor, we limit the inrush current to 4 amperes. It might take several seconds to

charge the capacitor to 400v at 4 amps but this is infinitely longer than the mere milliseconds it will take if the voltage is applied directly.

Of course, even one ampere of current at 400 volts is the equivalent of 400 watts of power. And so we need a large resistor capable of absorbing that amount of power without failure of itself. Since the precharge is only a few seconds, it doesn't need to be 400 watts, a 100 watt resistor would likely do. But the common resistor is typically ¼ watt, and would cinderize instantly. A power resistor is necessary.

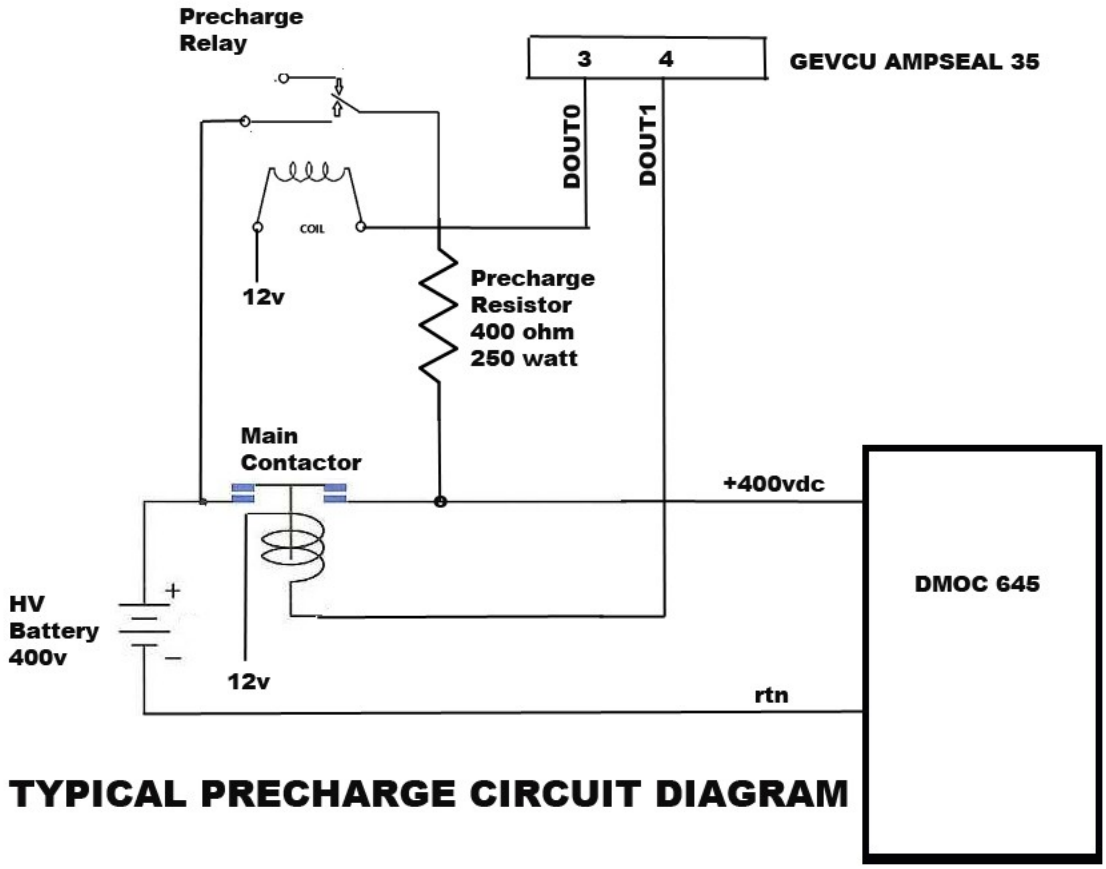
Unfortunately, once the caps are charged and the power switching begins, we don't want ANY resistance in our source voltage. As the switches often do 300 or 500 or even 1000 amps, it would give off a lot of heat and ultimately burn up even the largest resistor.

Once the capacitors are charged, we can safely connect them directly.

And so the usual process is to use TWO relays, your main contactor and a separate ***precharge*** relay that connects the power resistor across the main terminals of the contactor bypassing it.

And so the precharge process becomes:

1. Close precharge relay applying voltage to input capacitors.
2. Wait until capacitor reaches pack voltage.
3. Close main contactor connecting input capacitors directly to pack.



TYPICAL PRECHARGE CIRCUIT DIAGRAM

In the diagram, note that we use DOUT0 and DOUT1 to activate the precharge and main contactor relays respectively. ALL GEVCU digital outputs are a switched ground MOSFET. When set to 0 it presents an open on the associated AMPSEAL pin. When set to 1, it switches the MOSFET ON which connects the pin to ground.

The other end of both relay coils is connected to the ordinary vehicle 12v. So when the DOUT MOSFETS are turned on it provides a ground to the coil and closes the relay. These MOSFETS are capable of 2.7 amperes of continuous current and surge currents of up to 7 amperes – sufficient for the largest contactor coil.

Precharging is so common in these situations, that there are several variables already programmed into GEVCU software to accommodate precharging. To set these, bring up the serial terminal on the GEVCU and enter the following values:

PREDELAY=1500 This sets the precharge time delay in milliseconds

Setting Precharge Delay to 1500 milliseconds.

PRELAY=0 This sets the precharge relay output to DOUT0 on pin 3.
Setting Precharge Relay to 0.

MRELAY=1 This sets the main contactor relay to DOUT1 on pin 4.

Setting Main Contactor relay to 1

These values will be saved to EEPROM automatically. At any time in the future, when the GEVCU receives 12v on the input and completes its bootup process, it will immediately set the precharge relay output **DOUT0** on pin 3 to on, engaging the precharge relay and applying the 400v through the precharge resistor to the DMOC645.

The program will hold that state for the amount of delay you enter in **PREDELAY** in milliseconds. 1500 ms for example is 1.5 seconds.

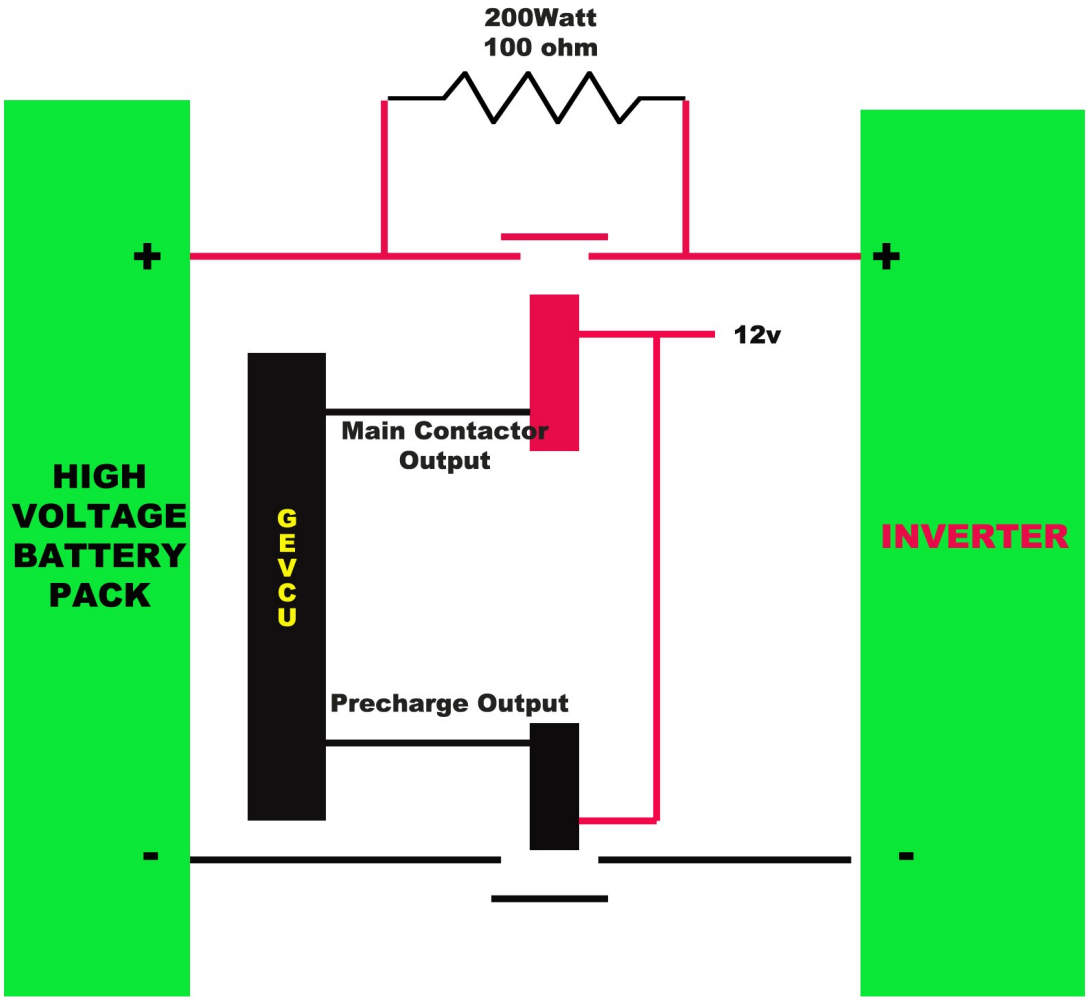
After that time has expired, it will close the MRELAY output – in this case DOUT1 on pin 4.

Note that any output can be used for precharge DOUT0 through DOUT7 and any output can be used for the main contactor.

If no precharge is desired, simply set **MRELAY** and **PRELAY** to 255. You can then use those outputs for other purposes. Note also that no precharge is done unless a PREDALY greater than 0 is entered.

AN ALTERNATE PRECHARGE TECHNIQUE

Precharge as depicted above requires a second relay and the usual approach is a much smaller relay to minimize expense – since it is only going to carry 3 or 4 amperes this appears to make sense. And on low voltage systems, it more or less works. But as we increasingly go to higher voltage AC drive systems, it more or less doesn't. The reason is contact arcing. The little 12vdc automotive relays are simply not designed to switch 350 vdc even at low currents. The high voltage potential itself will cause the contacts of these relays to arc – causing pitting and wear and potentially even welding them closed.



Simplified Two Contactor Precharge Circuit

The simplified two contactor precharge circuit shown here has several advantages for high voltage systems. First, it provides a redundant safety element as two high voltage contactor relays are used and both must be engaged to power the system. That means in an emergency shutdown scenario you have TWO contactors breaking the circuit. If one welds closed, the other one may indeed break the circuit.

But it also eliminates the use of the inexpensive 12v relay from the circuit. Note that one contactor is placed on EACH pole of the battery connection and a

precharge resistor is permanently wired across the terminals of the positive contactor.

In order to precharge the inverter, the contactor on the NEGATIVE pole is closed, completing the circuit through the resistor and applying pack voltage to the inverter through that resistor.

Once the precharge delay has been accomplished, the main contactor on the positive pole is then engaged – effectively taking the resistor out of the loop by bypassing it directly.

In this way, we can avoid the use of small failure prone relays in our precharge circuit. As of version 4.02 of GEVCU, we always leave the precharge output on after the main contactor is closed in order to provide for this type of precharge configuration.

When the GEVCU is powered down, both contactors open, removing all possibility of charge on the inverter.

7 THROTTLE CALIBRATION AND MAPPING

The concept of a “vehicle control unit” can cover a multitude of sins. But the most central issue, certainly with electric vehicles, is “controlling” the amount of power applied to the electric motor and drive train.

Early vehicles used a simple switch to apply power to the motor or not. By switching it in and out, the driver could kind of/sort of control the forward motion of the vehicle.

Later versions featured large potentiometers that consumed much of the power as heat but provided variable voltage to the motor.

Various switching schemes were used to switch batteries in various combinations to get different voltages to the motor.

But the objective is always to give the driver control of the forward motion of the machine.

In modern electric vehicles, we need to use the accelerator or throttle (we avoid calling it the gas pedal actually) to control the application of power to the electric motor to go forward. In most AC polyphase systems, this is complicated a bit by the fact that the AC drive motor can be used as a generator when coasting or slowing down. This is generally referred to as regenerative braking or simply “regen”.

Regen doesn’t just happen. We can control the AMOUNT or degree of power generation for any given turn of the wheels.

Most modern vehicles feature both forward acceleration control AND regenerative braking control using the throttle. Most drivers quickly become accustomed to this and learn to use it to great advantage with one pedal control right up to a full stop at the stop light. Some drivers do NOT like this feeling and do not want regen on the throttle at all.

The result is that the CENTRAL function of GEVCU is to translate throttle pedal position into motor control signals. It is also the most complicated concept to grasp as we need to accommodate as many combinations of pedal position, acceleration, and regenerative braking as possible to accommodate a variety of vehicles and driver preferences.

You will find that "tuning" the throttle map is THE most effective way available to tune the "feel" of your electric car when driving. And that driving "feel" is part of the magic of electric drive. Depending on the throttle map settings, your vehicle can provide a seamless interface where you actually feel like part of the car and control it effortlessly and without conscious thought. Done poorly, it can render a barely controllable vehicle prone to parking lot accidents and uncontrolled accelerations – a hazard to the driver and everyone on the public right of way.

HOW GEVCU DETECTS THROTTLE POSITION

GEVCU detects throttle position by voltage. Throttles may be potentiometers, hall effect devices, and of either one signal or two. GEVCU provides these sensors with a 5v supply and a ground reference return and receives from it a signal or signals indicating pedal position by voltage.

This signal is optically isolated, buffered, filtered, and scaled from the 0 to 5 volt signal from the pedal to a 0-3.3v signal that can be read by the multiprocessor. This signal is then sampled by the Analog to Digitial Converter and presented as a number between 0 and 4096 digitally.

So a 0v signal would produce a digital output of 0 and a 5v signal from the throttle would produce a value of 4096.

This is not a precise art. Sampling issues, noise on the wires, electromagnetic interference all have an effect on such small signals. So the software "averages" this input value continuously.

Programmatically, it then converts this digital value to a "throttle percentage" between 0 and 100 to some precision for use by the various program elements.

And so for example, the throttle at rest might put out a voltage of 0.83 and that is scaled and converted to a digital value of 95 which is then defined in software as 0% throttle.

With the pedal fully depressed, the throttle sensor might put out a voltage of say 4.62 volts. This is scaled, buffered, filtered, and converted to a digital value of 3785 and we then define that as 100% throttle.

THROTTLE TYPES

GEVCU rather arbitrarily divides the world of throttles into two types, single input and dual input. This is defined by the variable **TPOT** with **TPOT=1** indicating a single input throttle and **TPOT=2** indicating a dual signal throttle.

IF TPOT is set to 2, there is another consideration. Does the 2nd signal vary in voltage in the same direction to the first signal or is it inverted. **TTYPE=1** for a linear relationship and **TTYPE=2** for an inverted one.

THROTTLE CALIBRATION

It's an imperfect world. Whatever sensor is selected or used, the output read at the analog input to the GEVCU is going to vary depending on wire lengths, manufacturing variations etc. So it is important to have a means of calibrating the throttle.

For single input throttles two variables are provided – T1MN and T1MX. You simply enter the digital values read by GEVCU while the pedal is at rest (T1MN) and fully depressed (T1MX). We assume it will be more or less linear between those two points.

For two wire throttles, a second input is provided, T2MN and T2MX. If a single input is used, we want to set these values to zero.

To get the numeric values, we have to use our USB serial port terminal program to view the digital value "read" by GEVCU for the throttle positions. Enter the letter **L** and the serial monitor will begin to provide a text read out that is updated every few seconds.

Motor Controller Status: isRunning: false isFaulted: false

AIN0: 81, AIN1: 81, AIN2: 87, AIN3: 77

DIN0: 0, DIN1: 0, DIN2: 0, DIN3: 0

DOUT0: 0, DOUTO1: 0, DOUT2: 1, DOUT3: 0,DOUT4: 0, DOUT5: 0, DOUT6: 0, DOUT7: 0

Throttle Status: isFaulted: false level: 0

Throttle rawSignal1: 81, rawSignal2: 80

Brake Output: -200

Brake rawSignal1: 87

This display gives us quite a bit of information. Note the line AIN0: etc. This lists the actual digital values derived from our four analog inputs 0-3.

By convention, we usually wire the primary throttle signal to AIN0 and the secondary to AIN1 with brake inputs on AIN2.

The next line actually indicates the current status of digital inputs 0 through 3 as DIN0: etc.

And the next line will provide information on the current status of the eight digital MOSFET outputs DOUT0 through DOUT7.

So calibrating the throttle becomes quite easy to do manually:

1. Enter **L** to get the screen logging shown above.
2. Ensure throttle is at rest (idle).
3. Note numeric value appearing in **AIN0**.
4. Press throttle to full acceleration.
5. Note new numeric value appearing in **AIN0**.
6. Enter the first value using the command **T1MN=xx** where **xx** is the value displayed.
7. Enter the second value noted using the command **T1MX=xx** where **xx** is the value displayed.

For dual signal throttles, this procedure can be repeated for **AIN1** using the variables **T2MN** and **T2MX**.

THROTTLE MAPPING

Throttle calibration simply establishes the “end points” of the throttle map. We establish the actual digital readings for an idle and max throttle condition. This is then mapped to a normalized 0 to 100% throttle value.

GEVCU actually provides a rich set of variables to map that 0-100% in an almost endless number of combinations to “tune the curve” or tune your throttle action to vary the feel of the car while driving. The basic list of variables includes

TCREEP – amount of forward torque applied at idle. This can be useful with automatic transmissions for instance or to duplicate the feel of an automatic transmission ICE vehicle that wants to “creep” forward if you release the brake.

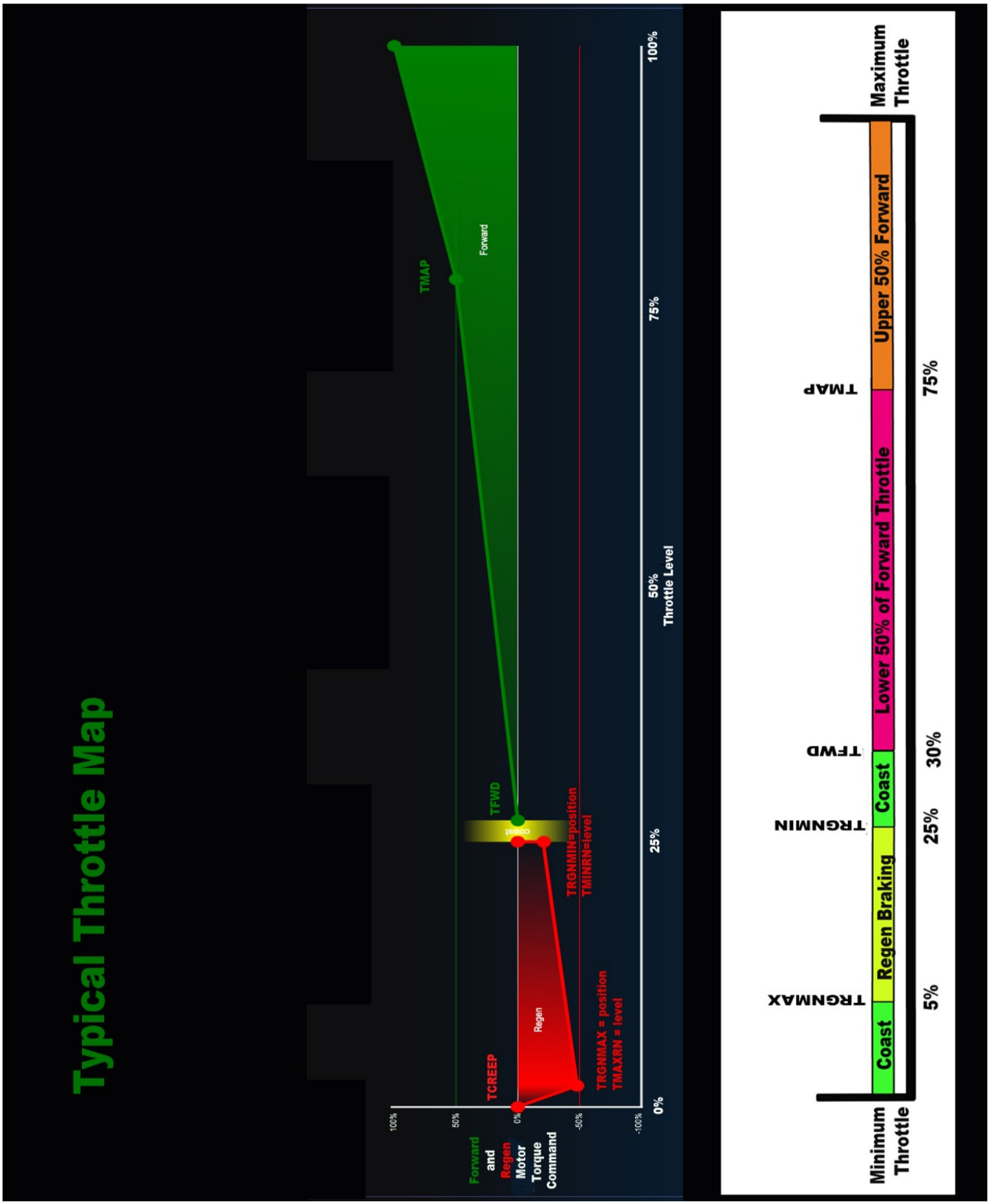
TRGNMIN – defines the point on the throttle where the minimum regenerative braking torque is applied. The minimum regen torque level is defined by an associated variable **TMINRN** which is the percentage of full torque.

TRGNMAX is the pedal position where maximum regenerative braking is felt and this maxiumum is defined by **TMAXRN** which is a percentage of full torque.

TFWD is the point on the throttle map where forward drive torque is applied.

TMAP is the point on the throttle map where 50% of available forward torque is applied.

Refer to the throttle map diagram.


Typical Throttle Map
TMAP
Forward
TFWD
TRGNMIN=position
TMINRN=level
TRGNMAX = position
TMAXRN = level
TCREEP
Regen
Coast
100%
50%
0%
-50%
-100%
Forward and Regen Motor Torque Command
0%
25%
50%
Throttle Level
75%
100%
Minimum Throttle
Coast
Regen Braking
Coast
Lower 50% of Forward Throttle
Upper 50% Forward
TRGNMAX
TRGNMIN
TFWD
TMAP
Maximum Throttle
5%
25%
30%
75%

The first thing to understand about the throttle map is that regenerative braking is a function of the forward motion of the car, fed through the axles and transmission to the motor, is used to generate electricity.

No matter what we set our variables to, there is no regenerative braking if the wheels are not turning.

So in the diagram depicted, the first 25% of the pedal does NOT cause us to go in reverse. It is simply dead pedal.

In starting off, **TFWD** defines the point in the pedal where forward motion begins. In this example, perhaps at 27% throttle.

Torque is then mapped from 27% to 100% throttle with one modification. It is scaled nonlinearly by the variable **TMAP** which defines the percentage of throttle where 50% torque occurs.

This is a nice feature actually. We can define 27% to 85% percent of throttle for the first 50% of available torque, with the remaining 50% torque mapped from 85 to 100%.

Why would we want to do this? It dramatically EXPANDS our resolution and control on the lower half of the available torque curve. In this way, we can markedly improve “feel” at low speeds when parking and maneuvering.

Typically, in driving the difference between hard acceleration and REALLY hard acceleration doesn’t really matter very much. You’re kind of foot down or not for max power. But for minimum power, a finer level of control is an asset.

Regenerative braking comes into play when we are already rolling and we begin to back off on the throttle. The kinetic energy of the car keeps it moving forward and left to its own devices will roll quite freely some distance. But this energy can be recaptured by using the motor as a generator and feeding the current back into the batteries.

Generally, as you back off the throttle you want to gently begin applying a minimum level of regenerative braking and increase that amount the more you back off on the pedal.

TRGNMIN defines, in tenths of a percent, the position on the throttle map where regenerative just begins to be felt. The associated variable **TMINRN** defines the percentage of maximum torque that is called for at that point.

The maximum regenerative torque is defined by **TMAXRN** and the point on the throttle at which this occurs is defined by **TRGNMAX**. In this example, we get max regenerative braking at about 5% throttle.

And so the further you reduce the pedal, the stronger the regen effect is felt, slowing the car.

Note a couple of gaps. We have nothing defined at all from 25% our minimum regen point, and 27% , the beginning of forward acceleration torque. This provides a little "coast" zone where no forward torque is applied and no regen torque is produced. You can make this zone as wide or as narrow as you like or not have one at all. I personally quickly find this "notch" in the pedal and often use it to coast freely down a hill for example.

Note too that **TRGNMAX** comes in at about 5% throttle. From that point to full idle is a second "coast" zone where no regen is produced and no forward torque either. Again, a matter of personal preference.

Again, **TCREEP** allows you to define a level of forward torque produced at 0% throttle.

Note that throttle input more than about 15% below the minimum you calibrated the throttle with will result in a fault condition and to about the same 15% tolerance any input ABOVE the **T1MX** will likewise result in a fault.

These variables are at first confusing and seem unnecessarily complicated. Worse, it will take you a number of "tuning drives" where you go drive the car, come back and change some variables, go drive the car, come back and repeat.

But you will quickly see that these variables allow you to achieve a custom level of feel and nuance in the way the vehicle drives that is really quite the central feature of GEVCU.

In other words, it is well worth the effort to achieve a truly pleasurable driving feel.

8 BRAKE CALIBRATION AND MAPPING

Brake calibration and mapping is very similar to that for the throttle, but much simpler as there is usually only one brake input.

Braking issues revolve around the concept of regenerative braking described in the Throttle section of this manual. With most AC drive systems, the traction motor can be used as a generator during deceleration converting the kinetic energy held in the forward motion of the car into electrical current, which is in turn fed back into the batteries to “restore” a bit of energy.

In theory, this sounds like we can recover a significant amount of energy and extend our range. In almost all of our actual comparison tests, we’ve found the realized energy efficiencies of regenerative braking in all cases much less than advertised, and really quite meager when demonstrated. The reasons for this are a bit complex, but involve the fact that WITHOUT regenerative braking, you drive the car quite differently and quickly learn to take advantage of “free roll” which is a very different feel from normal ICE vehicle operation. In an internal combustion engine, when you take your foot off the throttle, the compression of the engine acts as a kind of a brake. When you remove your foot from the throttle of an electric car, an electric motor HAS no engine compression and so does not. As a result, you can often roll for a long distance using no electricity at all.

If we measure energy OUT of the battery when accelerating, and energy IN to the battery during regenerative braking, there APPEARS to be a considerable energy savings. But without regenerative braking, we find we drive the car quite differently, and the energy OUT to drive the car changes considerably. As a result, comparing the same actual drive WITH regenerative braking and without regenerative braking simply does NOT show those kidns of efficiency gain.

So why use regenerative braking at all? It actually puts more load and stress on the motor, the inverter, and indeed the batteries. It’s a good question.

But over time, we’ve learned we just LIKE the feel of regenerative braking and it DOES reproduce the feeling of engine compression in a more conventional car feel. Even better, we soon learn to use just the throttle to modulate between forward acceleration and regenerative braking giving us a new kind of one foot control of the vehicle that many electric car enthusiasts strongly prefer.

With GEVCU You can easily turn regenerative braking off entirely for either or both the throttle and brake. But you can also use regen for either or both the throttle or the brake.

Regenerative braking on the brake gives us a sense of power brakes. We usually tune this so that just a little brake pressure will give us quite a bit of regenerative braking pressure and this increases with pedal pressure up to a point. But at the point where the mechanical hydraulic brakes actually come into play, we often cut off the regen entirely. In this way, a light brake pressure gives us some regen deceleration, heavier foot pressure mostly giving us just mechanical braking.

To use regenerative braking on the brake, you will need some sort of 5v sensor indicating brake pedal travel. We actually strongly prefer using a hydraulic brake pressure transducer that provides a 0-5 output based on hydraulic pressure felt in the actual brake lines. This sensor usually has three connections, 5v, GND, and the output signal. GEVCU provides several 5v and GND outputs to choose from. From there, it is an easy matter to tie the brake pressure transducer output signal line to one of our analog inputs. By convention, we use the first two analog inputs, 0 and 1 for the throttle. And so we normally use input 2 for braking.

Note that regenerative braking on manual brake systems can be quite pleasant to use. Generally using regenerative braking on power brake systems is of much lower usefulness in tuning a car for ideal braking feel.

Calibrating the brake becomes quite easy to do manually:

1. With a terminal program, bring up the serial port screen display.
2. Enter **L** to get the screen logging shown above.
3. Ensure brake pedal is at rest .
4. Note numeric value appearing in **AIN2**.
5. Press brake pedal to maximum.
6. Note new numeric value appearing in **AIN2**.
7. Enter a value JUST ABOVE the first value noted with the brake off, using the command **B1MN=xx** where **xx** is the value displayed.
8. Enter a value JUST BELOW the second value noted using the command **B1MX=xx** where **xx** is the value displayed.

.This sets the minimum and maximum input levels GEVCU will see on AIN2.

BRAKE MAPPING

Brake calibration simply establishes the "end points" of the brake map. We establish the actual digital readings for an off brake and max brake condition. This is then mapped to a normalized 0 to 100% brake value.

Two additional variables are used to establish the level of regenerative braking to be used. **BMINR** represents the minimum level of regenerative braking we want to exhibit while braking while **BMAXR** establishes the maximum degree of regenerative braking . The valued entered will indicate PERCENT of available regenerative braking torque to be applied. Values of 0 will turn off regenerative braking on the brake pedal.

In applying regenerative braking, GEVCU will map **BMINR** to **B1MN** to give minimum regenerative braking when you just start to press the pedal. It will linearly apply regenerative braking up to **BMAXR** occurring at the pedal position you identified as **B1MX.**

BRAKE LIGHTS

One of the advantages of ac induction motors and polyphase brushless dc motors is that it is very easy to switch roles from being a drive motor to acting as a generator. Indeed, when you remove power from the motor but continue to turn its shaft from the forward motion of the vehicle, it basically reverts to acting as a generator.

We can control this with our inverter determining how much power is produced, and thus how much of a slowing effect it will have on our car. Indeed we provide variables in the GEVCU to allow you to tune how much regenerative braking occurs when you put on the foot brake, and indeed how much occurs when you back off the throttle and specifically where on the throttle that occurs. In fact, an inordinate amount of the setup of GEVCU is devoted to this one issue – regenerative braking.

This gives rise to an anomaly you may want to consider. When you put your foot on the brake of most automobiles, an electrical switch closes turning on the brake

lights on the rear of the car. This switch is usually located on the master cylinder and reacts to hydraulic pressure, but on some vehicles it is actually on the brake pedal itself. In either case, when you start to brake, two big red lights on the rear of the car, required by the National Highways Traffic Safety Administration, light to warn drivers behind you that you are slowing down.

This is an important safety feature to prevent large trucks from cohabitating in your trunk with your spare tire.

The issue is with throttle-based regenerative braking. Most electric vehicle builders and drivers eventually discover a pleasant level of control of the vehicle that can be obtained using regen on the throttle. They quickly learn they can slow almost to a stop just using the accelerator and almost never have to take their foot off the throttle to brake. Indeed, one of the ways you can spot an EV conversion is to look for the rusty brake disks. Some EVs simply do not ever have to have new brake pads because the braking system is almost never used.

While regen on the throttle can be used to dramatically slow the car, it doesn't inherently light the brake lights. And so some EV drivers find they always seem to have people climbing their tailgate. The reason is the other drivers were not provided sufficient warning of your decreasing speed during regenerative braking.

GEVCU makes provisions for this with the BRAKE LIGHT output. You can designate any of the 8 digital outputs as a brake light output. The system will monitor the actual torque output reported by the inverter to the GEVCU via CAN. In any event where this torque is NEGATIVE and exceeds 10 Newton Meters, the brake light output is set to ON.

Via serial terminal, this value is set with the BRAKELT variable.

BRAKELT=5 This sets the brake light output to output 5.

Brake light output updated to: 5

If you do not need a brake light output, set this value to **255**.

This variable is also available on the wireless website configuration page.

Like other digital outputs, this is a switched MOSFET ground. When the torque value goes negative to more than 10 Newton Meters, the output pin is grounded. Otherwise it is open.

You can easily use this output to provide a ground to the coil of an ordinary automotive 12v relay and use the relay to bypass the brake light switch on either pedal or master cylinder to switch on the brake lights just as if the brake pedal had been pressed.

In this way, when slowing through regenerative braking, you WILL light the brake lights on the rear of the car, warning drivers behind you that you are slowing.

The 10 nm threshold requirement allows modest regen without tail lights but any substantial regen above this modest threshold will light the tail lights.

9 POWER VALUES

There are a couple of variables having to do with maximum power and rpm for the motor overall. These can be set using the USB serial port and terminal program or the wireless web server.

Torque is a measure of pressure or work of a rotating shaft. It is normally expressed in the United States as ft-lbs and in the rest of the more metric world in Newton Meters. 1 newton meter = 0.737562149 foot pounds. Conversely 1 foot pound = 1.35581795 newton meter.

There are two ways to operate power switching control of AC motors. You can command them to a speed, using all available torque at a certain ramp rate, or you can command torque itself and largely ignore speed. For most automatic operations such as operating pumps and compressors, you would use speed control. You want to set a certain speed and have the motor run at that speed and it will take whatever torque is necessary to reach that speed and maintain it.

But for electric vehicles, we normally use torque control. Your foot commands a certain amount of torque and the speed is left mostly up to the driver to monitor on the speedometer. When starting from a stop, you need maximum torque to accelerate but as you reach your desired speed, you naturally come off the throttle decreasing the applied torque. You don't really care what speed the motor shaft turns out to get there.

The **TORQ** variable establishes the maximum level of torque that GEVCU will command the DMOC645. The combination DMOC645 and Siemens Motor may or may not actually achieve this depending on available battery voltage, gearing, etc. But this is the maximum value, expressed in tenths of a Newton Meter, that GEVCU will request. The DMOC645 actually reports back the ACTUAL value of torque at any one instant.

We think of DMOC645 and Siemens motor as capable of up to 300 Newton Meters of torque. So for maximum torque with this combination, you would enter the following serial command:

TORQ=3000

Note that this is in TENTHS of a Newton Meter and so the value 3000 represents 300 NM.

A second value is established programmatically in a configuration file to set the maximum regenerative braking torque. No idea why this wasn't made an accessible variable, but it will typically be set to 40% of whatever the maximum **TORQ** value is.

Other variables specific to the brake and throttle actually establish what percentage of THAT value Is applied at specific brake or throttle levels.

It is usual to set a maximum rpm level for the electric motor. Regardless of WHAT level of torque you command, the DMOC645 won't apply it if this maximum revolutions per minute level is achieved. This is set by the command

RPMS=6000

For example this command would set the max rpm at 6000 rpm.

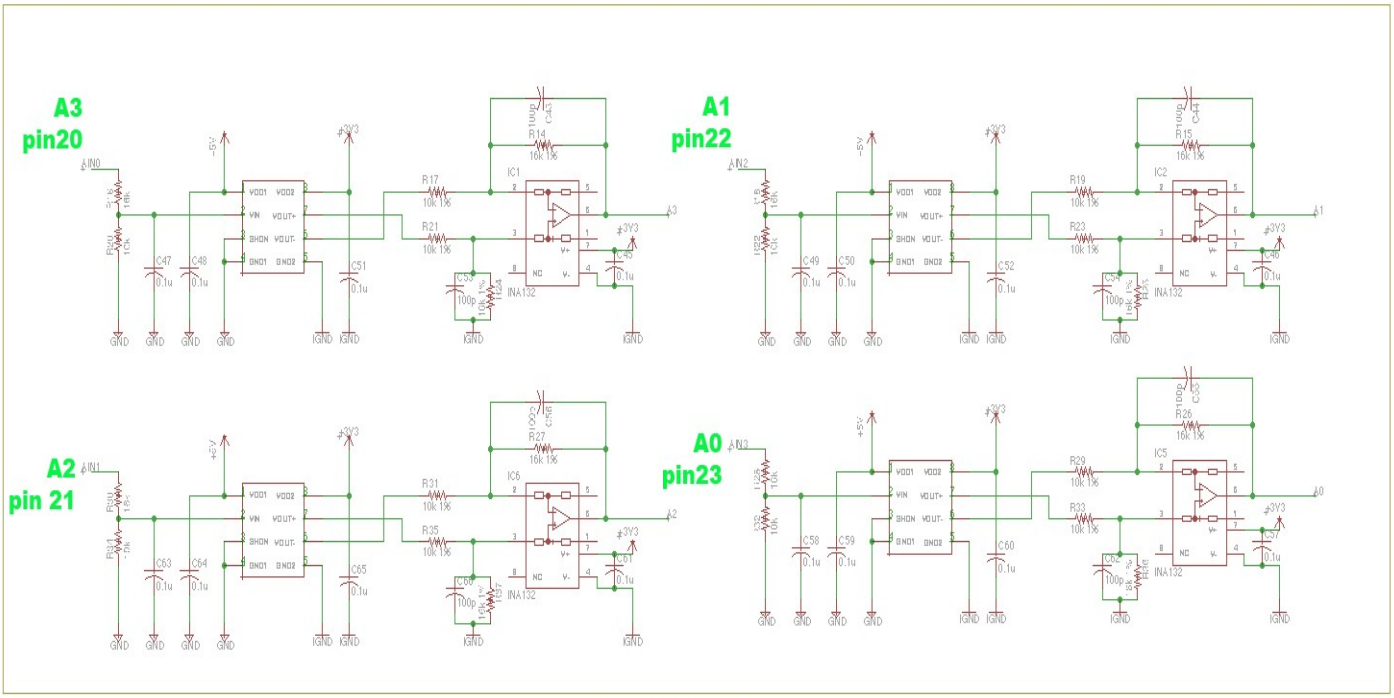
An odd variable included is the percent of torque available when in REVERSE. The DMOC645 does not actually know when you have placed the transmission in reverse unless you use one of the inputs to note that and wrote software to do this. But there are some provisions for using a digital input to put the motor direction in reverse. And this variable would establish a separate maximum torque when backing up using the reversed electric motor. This is expressed in tenths of a percent.

REVLIM=500

This command would limit maximum torque WHEN in reverse to 50% of maximum available torque.

The **TORQ** command can actually be quite useful. For example, by setting **TORQ=750**, you might limit the motor output to 75 NM. You can then map your throttle and brake and test drive them without much of anything getting away from you because you have strongly limited the power output of the motor. Once your braking and throttle feel right, even though the vehicle is a little docile, you can then return **TORQ=3000** for 300 NM of torque for full power.

10 Analog Inputs



ANALOG INPUTS

Since the central role of the GEVCU is to allow control of the power applied to the motor as directed by the throttle input, we have invested a serious amount of design in the four analog inputs to the microprocessor.

The SAM3X microprocessor features 12 bit analog to digital conversion inputs. But these inputs are limited to the processor operating voltage of 3.3volts which poses a bit of a problem as almost all automotive sensors are 5v devices. This A/D input works by sampling the input to produce a digital numeric value between 0 and 4096 representing a voltage of 0 to 3.3v. In an ideal circuit this represents a precision of 0.8 millivolts per digit.

In order to protect the microprocessor from the harsh noisy environment of automotive wiring, we need to optically isolate the input – essentially removing any direct electrical connection.

We do this with a bit of a pricey but very capable chip from Avago Technologies – the ACPL-C87. This is a Precision Optically Isolated Voltage Sensor and it features a unity gain, a very high linearity but an operating range of 0-2 volts.

We first divide the 5v input using a 16/10k voltage divider to produce a 38.5% signal ratio. This gives us a 1.92 volt input for a full 5v signal which is just under the 2.0v maximum input for the chip. A series of filter capacitors act to remove noise spikes both from the input line and the local power supplies at the chip location.

This signal of course drives a light emitting diode. An associate photo transistor then conducts in a very linear fashion to the amount of light received and this chip, rather than simply passing digital ON or OFF signals, will actually produce an output directly proportional to the applied input voltage. Indeed it will produce a differential output that effectively cancels any noise spikes that may have made it through as well.

The differential output is fed to a more common operational amplifier producing a single output representing the difference between the two signals from the Avago chip. The gain of this opamp is set again to scale the output appropriately to the 0 to 3.3v input to the SAM3X microprocessor.

The GEVCU also provides a series of 5v and GND connections at the AMPSEAL 35 connector. These can be used to power and reference the external brake and throttle sensors.

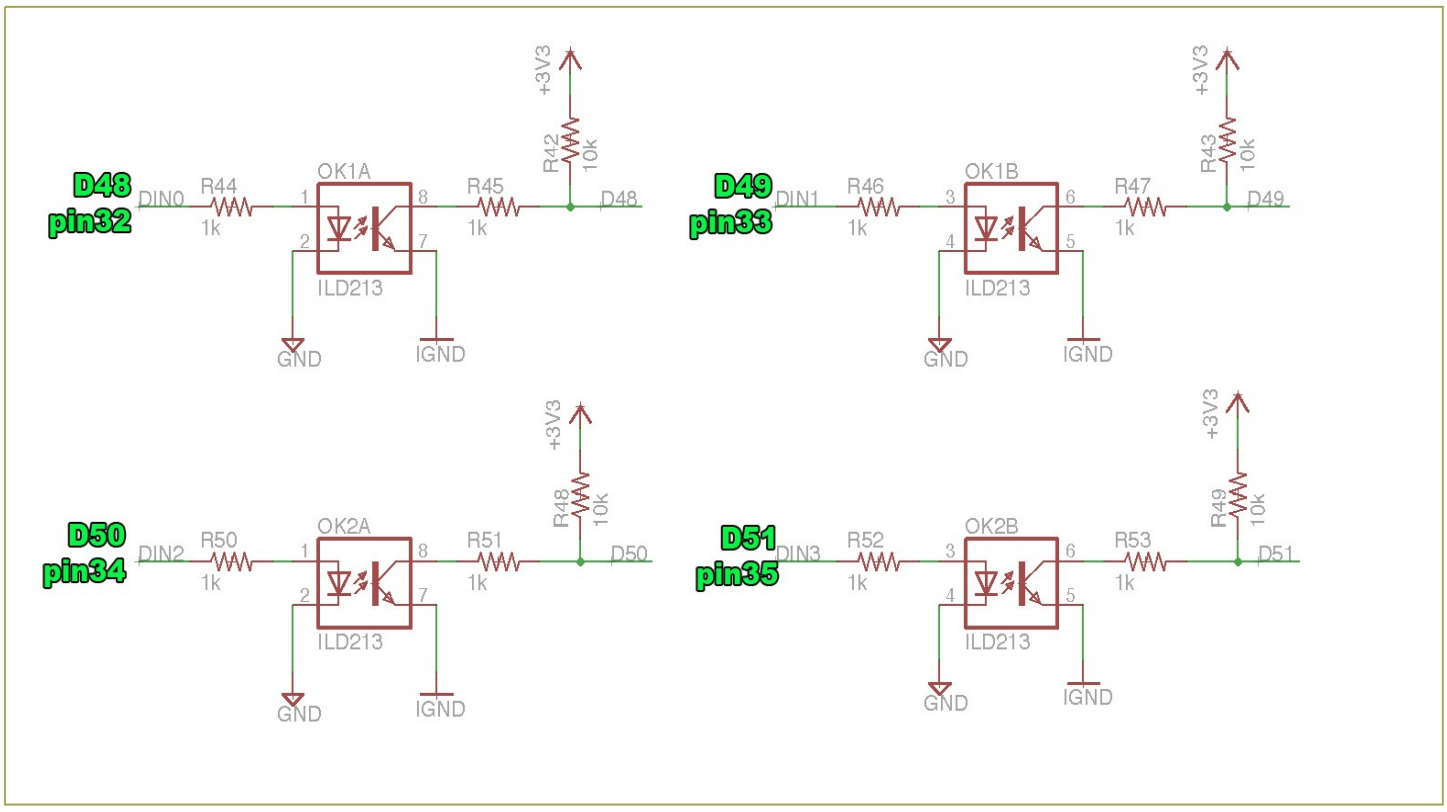
In this way, the GEVCU can monitor four analog signals and convert them to digital values, but the microprocessor is entirely optically isolated from external components and appropriate noise suppression is provided at the same time. This should provide stable and safe analog input readings.

GEVCU ANALOG INPUT	WIRE COLOR	AMPSEAL PIN	ARDUINO INPUT
AIN0	ORN/BLK	20	A3
AIN1	PNK/BLK	21	A2
AIN2	TAN/BLK	22	A1
AIN3	YEL/BLK	23	A0

When specifying analog input pins to GEVCU software the table above maps the GEVCU Analog Input 0 through 3 to the corresponding AMPSEAL pins and Arduino InputS. In the serial port configuration software, simply use the digits 0, 1, 2, or 3 to specify which input you intend to use.

So far, by convention we have used inputs 0 and 1 for throttle, input 2 for brake and input 3 is available.

11 Digital Inputs



DIGITAL INPUTS

The GEVCU provides four optically isolated digital inputs. By digital, in this case we are not referring to TTL level inputs but rather switched 12v inputs that can be read and acted on.

A 1kohm current limiting resistor allows about 12ma on the input to an ILD213 opto-isolator chip. This lights an LED which puts a phototransistor into conduction.

The output uses a 10k pullup resistor to the isolated 3.3v supply of the SAM3X microprocessor to present a TTL high on the microprocessor input pin which is driven low by the input signal which switches the transistor into the on state. A 1k current limiting resistor on the output causes about 0.3v on the microprocessor input which is read as a low. From the perspective then of the GEVCU software, these inputs are active low – essentially inverting the 12v on 0v off input.

GEVCU DIGITAL INPUT	WIRE COLOR	AMPSEAL PIN	ARDUINO INPUT
DIN0	ORN/BLK/RED	32	D48
DIN1	PNK/BLK/RED	33	D49
DIN2	TAN/BLK/RED	34	D50
DIN3	YEL/BLK/RED	35	D51

The table above shows the GEVCU digital inputs, associated AMPSEAL35 pin number and the Arduino Due digital input equivalent. When specifying digital inputs on serial port configuration items, simply enter the DIN number 0, 1, 2, or 3.

FEATURE EXAMPLES

ENABLE SIGNAL

Normally, when 12v power is applied to the GEVCU, it goes through an initialization and precharge procedure and when this is completed it is up and ready to drive the inverter and motor.

We can revise this operation to allow GEVCU operation on power up, but keep the motor/inverter disabled until we actually “turn it on” with a 12v enable signal.

This is done via serial port with the **PINENABLE** command.

PINENABLE=2 will set digital input DIN2 as the ENABLE input. When 12v is present on AMPSEAL pin 34, the motor inverter will be enabled and when 12v is removed at any time, it will be DISABLED.

This enable function is ONLY active if **PINENABLE** has a value of **0, 1, 2**, or **3**. If you do not need this feature, set **PINENABLE=255**.

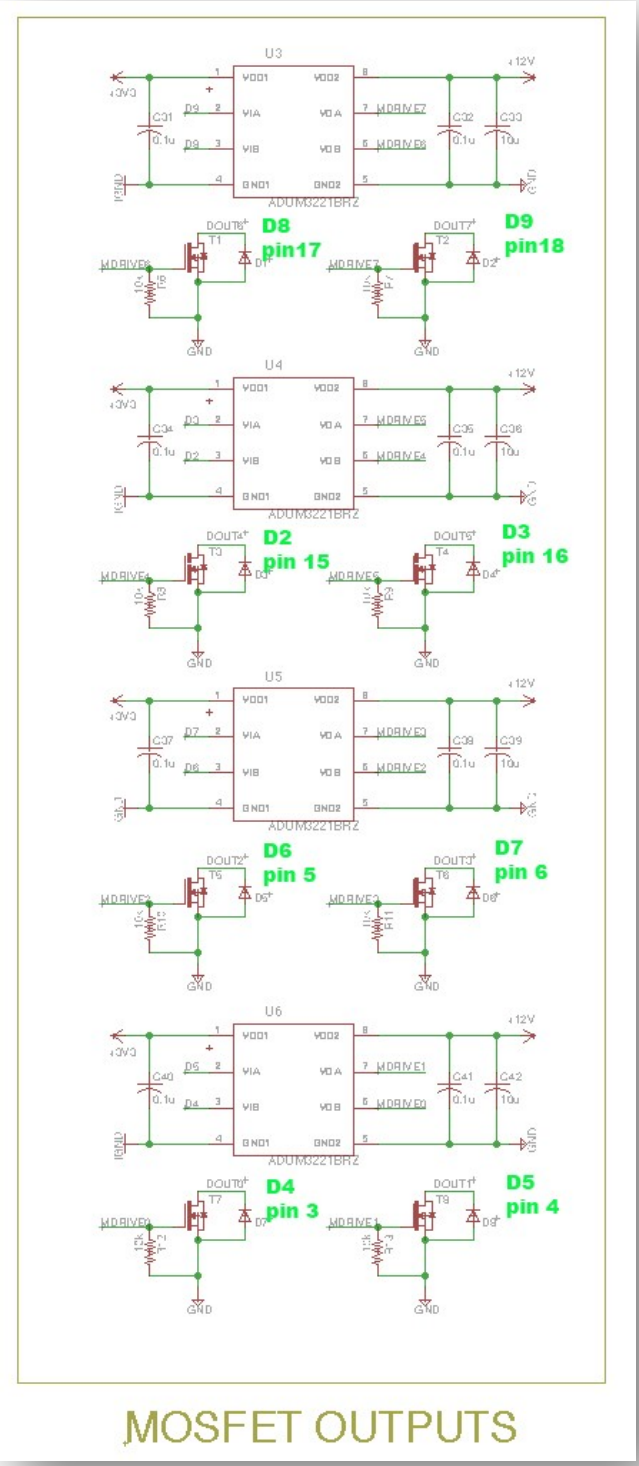
REVERSE INPUT

The default mode of the GEVCU is to turn the motor in the forward direction only. But we can use one of our digital inputs to allow both forward AND reverse operation of the motor by setting a **PINREVERSE** input.

PINREVERSE=2 will set digital input DIN2 as the reverse input. When 12v is applied to AMSEAL pin 34 then, the drive signals to the motor will cause the shaft to spin in the REVERSE direction. When the 12v is removed, it will revert to FORWARD operation.

This reverse function is only active if **PINREVERSE** is set to **0, 1, 2,** or **3**. To disable the reverse function, set **PINREVERSE=255.**

12 Digital and Analog Outputs



With GEVCU version four, the number of digital outputs has been increased from four to eight.

The basic SAM3X microprocessor features many digital outputs which can each source up to 50 ma of current with a maximum of all outputs being further limited. This is inadequate power to drive even small relays or devices found in the automotive environment.

Additionally noise and inductive voltage spikes found on such automotive outputs are essentially instant death to the low voltage (3.3v) SAM3X microprocessor.

GEVCU takes the bare digital outputs from the microprocessor and buffers and isolates them using four Analog Devices Isolated Dual Channel Gate Drivers (ADuM3221). These gate drivers provide full isolation and outputs up to 4 amperes to drive MOSFETs or IGBTs.

The outputs of these gate drivers are used to switch Infineon Technologies BSP295 N-channel

MOSFET small signal transistors. These MOSFETs feature a very low forward resistance when ON (0.3 ohms) and can carry up to 1.7 amperes of current continuously with some 7 amperes pulse at voltages up to 50v. This is sufficient power to drive most relays and indeed the contactors we often use to switch high voltage high currents in electric vehicles.

The output is basically an active low. With a digital 1 or high on the output, the MOSFET is switched into conduction effectively connecting the output pin to vehicle ground. You would normally use an external pull-up resistor connected to 12v somewhere external to the GEVCU to power the output.

GEVCU DIGITAL OUTPUT	WIRE COLOR	AMPSEAL PIN	ARDUINO PIN
DOUT0	TAN	3	D4
DOUT1	VIO	4	D5
DOUT2	GRY	5	D6
DOUT3	YEL	6	D7
DOUT4	BLU/RED	15	D2
DOUT5	VIO/BLK	16	D3
DOUT6	GRY/BLK	17	D8
DOUT7	GRN/RED	18	D9

The table above lists GEVCU digital outputs 0 through 7, the associated AMPSEAL 35 pin, and the Arduino logical pin corresponding.

It is worth noting that in addition to activating relays, lighting LEDs or lamp bulb indicators, with appropriate software any or all of these outputs can also be pulse width modulated.

The SAM3X has the ability to convert a 12-bit digital value between 0 and 4096 into a PWM output on these pins using the AnalogWrite function. The pulse width of the square wave output would be a function of the output specified, with 0 indicating off and 4096 being constantly on. A value of 410, for example, would normally give a 10% duty cycle and on a 12v output that would be about 1.2v.

But do be aware that the active low configuration of the MOSFETS kind of inverts the PWM function. On is off, and off is ON so that the PWM actually varies in reverse with

4096 being off or GROUND all the time at the MOSFET and 0 being ON or 12v all the time. In this case, 4096-410=3686 for a 10% 1.2v analog output.

In this way, a simulation of an analog signal between zero and 12v can be produced on any of these outputs to drive legacy vintage temperature gages or fuel gages for example. As the frequency of this PWM signal can also be programmatically altered from the default 1000 Hz square wave, the analog simulation can be made to be really quite effective.

13 COOLING CONTROL – A DIGITAL OUTPUT EXAMPLE

One of the immediate needs for the DMOC645 and Siemens motor is for liquid cooling. If these devices exceed certain temperatures, the DMOC645 will limit the output current in an attempt to hold the temperature within operational limits. This is one of those very quiet “gotchas”. With inadequate cooling, your EV will operate JUST FINE. But you will be completely unimpressed with the power and acceleration of the Siemens Motor and DMOC645 controller.

With adequate cooling, this pair will put out nearly 300 nm of torque, which for anything under 3000 lbs with a transmission will feel very responsive. And so if you find yourself disappointed in the performance of this pair, your first examination should be to make sure you have adequate cooling for the system. In general, people grossly underestimate how much heat these devices give off and how much cooling is required.

On the VW Thing, we run the inverter and the motor on the same cooling loop using a Liang Solar pump which does about 17 liters per minute through the AN-6 braided nylon hoses. Cool liquid flows first through the DMOC645, which is quite sensitive to heat, and then the Siemens motor, which is somewhat less so. Well and good enough. But the trick is you have to REMOVE the heat from the system. We use TWO Denali heat exchangers with quite powerful fans on them to do this.

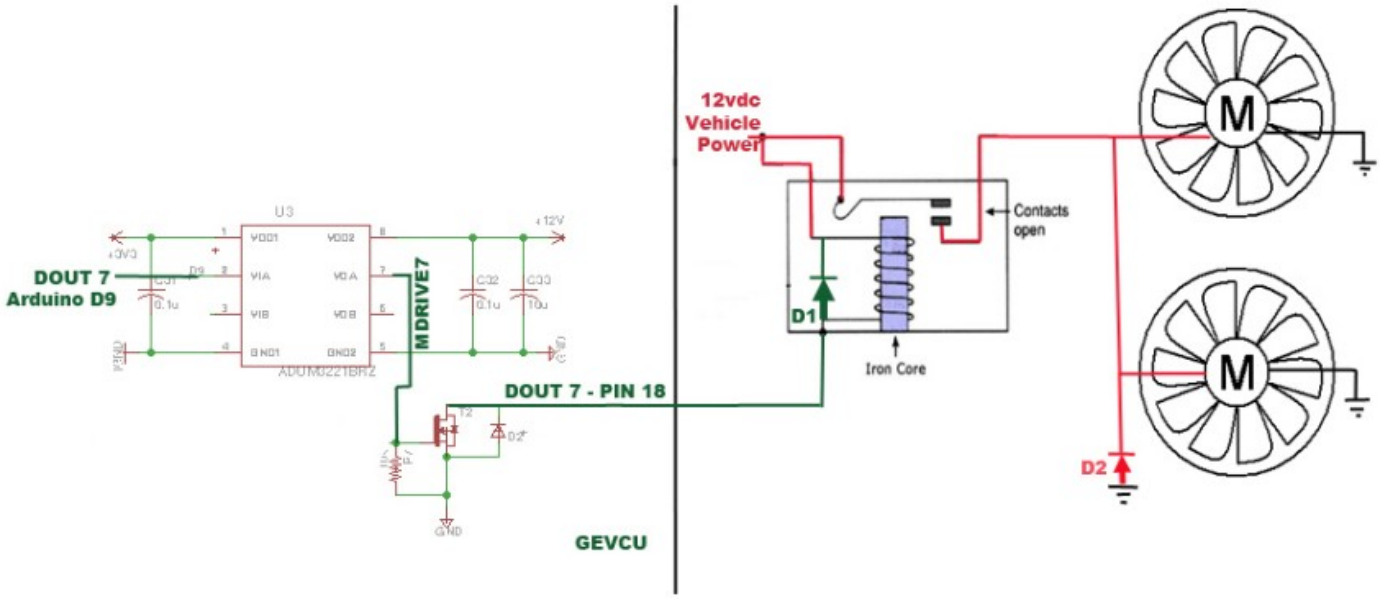
Of course, the fans not only use quite a bit of 12v energy, but they also are uncomfortably noisy. This is not so much a factor on the highway, but tooling around town it’s a little loud.

Oddly, tooling around town we do not NEED much cooling. The circulating pump and the heat exchangers operating without the fans works just fine. But if we DO go out on the highway and operate the motor and controller for more than 10 or 12 minutes, we will go into current limit. With the fans, adequate cooling is provided and we do not.

Similarly, July temperatures of 100F ambient don’t help our cause. But January’s 15F temperatures render cooling needs almost moot.

And so we have built in a cooling control function that gets Inverter temperature from the DMOC645 via CAN message, and compares it to set limits. If the temperature goes higher than the COOLON temperature we designate, we set DOUT7 to on to activate our fans.

If the temperature should later fall below the value we set as COOLOFF, the digital output is turned off, and so the fans are taken offline. In this way, we can use the fans on the Derali heat exchangers only when they are needed.



The diagram above shows the external connections to the GEVCU. When DOUT7 is set to ON, it causes the MOSFET driver to send a drive signal to the output MOSFET switching it into conduction. This is connected to pin 18 on the Ampseal connector which is routed to the ground side of the fan relay. 12vdc from the vehicle is connected to the other end of the coil AND to the common terminal.

When the MOSFET is turned on, this completes the path for the 12v through the coil, energizing the relay. This applies 12vdc to the two heat exchanger fans through the relay.

Note the use of diodes D1, across the relay coil, and D2, from the fans to ground. These are coil suppression diodes. When we turn cooling off, the coil in the relay will seek to continue the current flow. This can cause spurious voltage spikes that can rise to hundreds of volts and could even damage the MOSFET output circuit in the GEVCU. Diode D1 provides a path for current flow from the inductance of the coil and this energy is dispersed harmlessly.

The fans are of course driven by electric motors, but the fan blades themselves have inertia. Not only do the primary windings of the fans have high levels of inductance, but the kinetic energy in the fan blades will keep them turning briefly after the relay opens removing power. In this sense, they act like generators and again voltage spikes are the norm. Relay D2 provides a current path for those, shunting them harmlessly to ground.

There are three configuration variables in GEVCU that control cooling output, **COOLFAN**, **COOLON**, and **COOLOFF**. These can be set using the serial terminal.

COOLFAN=7 This sets the specific digital output to use for cooling (0-7).

Cooling fan output updated to: 7

COOLON=65 This sets the temperature at which the output goes active.

Cooling fan ON temperature updated to:65

COOLOFF=55 This sets the usually lower temp at which the output becomes inactive..

Cooling fan OFF temperature updated to: 55

You can avoid the fans cycling quickly off and on by setting **COOLOFF** somewhat lower than **COOLON**. In this way, the fans will come on and stay on for a while until the coolant temperature really comes down. It will then go off and remain off until the temperature again rises to the **COOLON** temp.

15 WIRELESS CONFIGURATION

One of the central design principles of GEVCU was that it be of course very configurable for a variety of vehicle applications WITHOUT requiring the user to learn to program software in C++. But further, we did NOT want to invent another device requiring another separate "configuration program" that was operating system dependent and constantly in need of upgrade. We required all configuration to be onboard, and operating system agnostic.

The basic configuration mode, already described, is via a serial terminal connection via the devices Universal Serial Bus (USB) connector. ANY device that can act as a serial terminal over USB can access this very rudimentary menu and enter modify any of the available configuration variables.

A SECOND way to configure GEVCU, again, with no external program or operating system, is via a wireless web site connection using any device that will do wireless connections and provide an HTML browser. Laptops, iPads, iPhones, Androids, we don't care what you use now or what you will use in the future. As long as Wifi and web browsers remain in vogue, you should be able to configure GEVCU.

The nature of online web sites is such that the configuration can be made more graphic and intuitive in this way.

GEVCU features an onboard WiFi 802.11b/g receiver transmitter with actually a full TCP/IP stack on it allowing bridge communications, ftp, http, e-mail and really a very capable set of other internetworking options barely taken advantage of in the GEVCU software. Programmers and software developers may note that for a full description of all that CAN be done with this very feature rich device, see the ConnectOne Programmer's Manual at

http://www.connectone.com/wp-content/uploads/2013/08/ATi_Programmers_Manual_8_41.pdf

For the GEVCU, we selected the ConnectOne module because it has its own multiprocessor and does all the work to display a web site allowing you to make configuration changes to the system without unduly loading the GEVCU multiprocessor. GEVCU's primary task and priority is of course realtime monitoring and control of the vehicles power control through the inverter.

Although we use a very fast ARM CORTEX 3 processor, we do not want to load it down with housekeeping chores that might detract from the time it devotes to this more important task.

So the ConnectOne processor actually manages the website. If you actually do make changes to the configuration, ConnectOne interrupts the GEVCU which then retrieves only the CHANGED variables very quickly and easily from the ConnectOne module without having to do ANY work on maintaining the web site or parsing data.

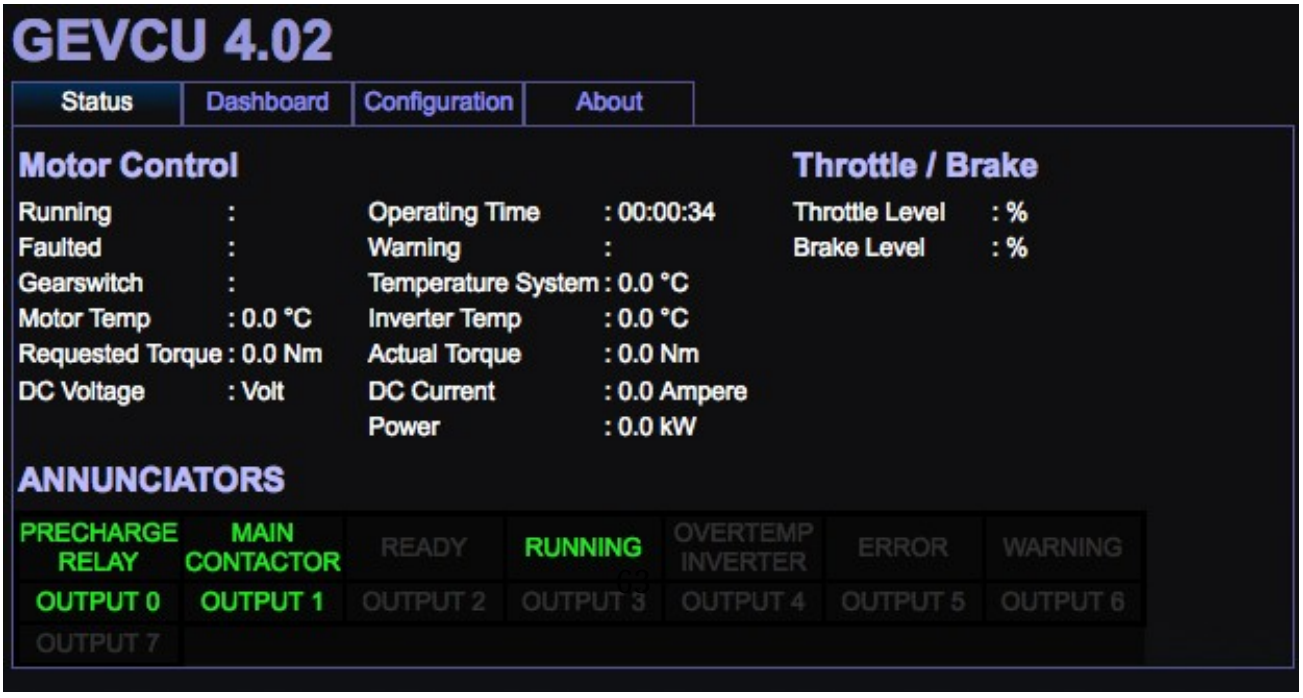
As a result, most of the changes you can make to GEVCU through the very Spartan serial port interface, can be made much more easily and wirelessly using a laptop and wireless connection to the GEVCU.

GEVCU sets up an AD-HOC 802.11b/g wireless presence with SSID **GEVCU**. So simply scan for a **GEVCU** ssid with your laptop wireless program.

Once connected wirelessly to GEVCU, you can access the web site using any browser program at http://192.168.3.10

You may be asked for a password. The default password is "**secret**". Use this same password when submitting changes to configuration.

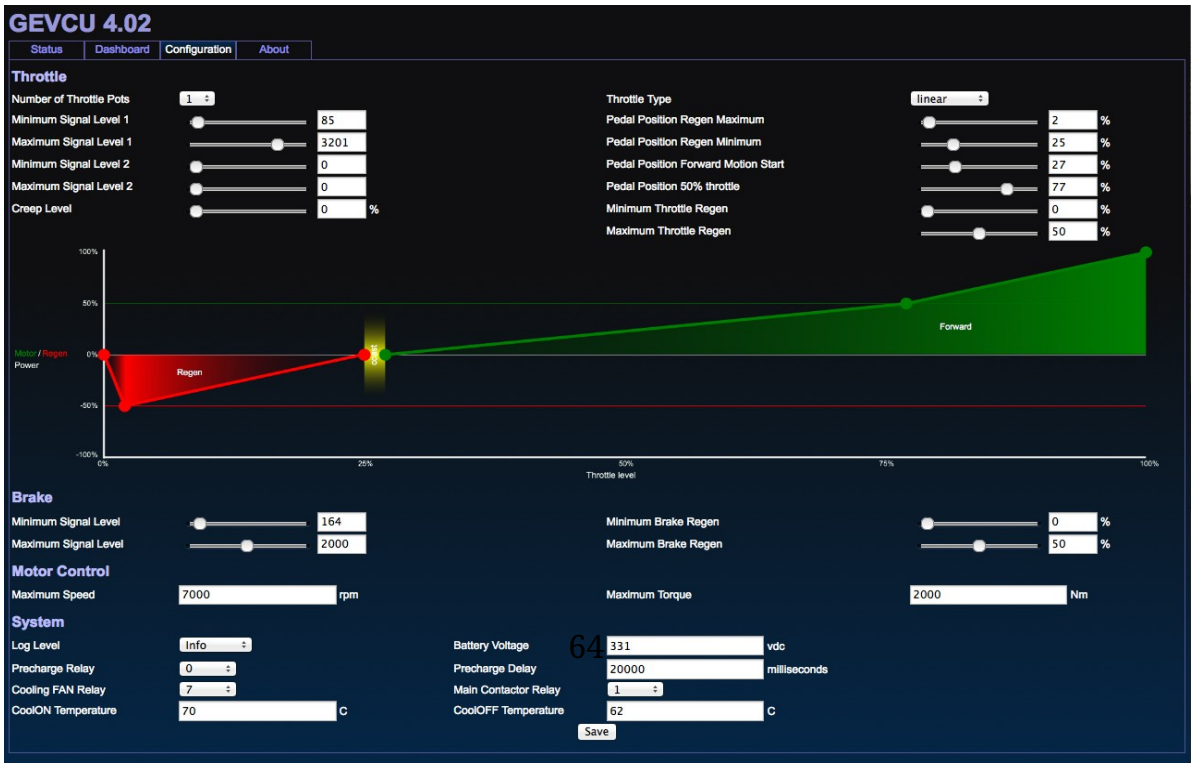
The first screen you should see is the STATUS screen. This screen shows various DMOC645 operating parameters in text form and provides a darkened panel beneath with certain error lights (shown illuminated) that come on in different operating status scenarios.



A related DASHBOARD screen shows live controller data in somewhat more graphic form.
The important screen is of course the configuration screen allowing you to set brake and throttle values and regenerative braking and forward acceleration values. This is slightly more intuitive than the serial port, although often you will

need the serial port L command to log actual sensor minimum and maximum values to be able to set this intelligently.

Note that you can also set maximum torque, rpm, and type of throttle here. As you make changes with the value sliders, you can see those changes reflected in the throttle map diagram.



Indeed most (but not all) of the available variables you can change to alter operation of the GEVCU are available on this configuration screen.

When doing this, note that it has no immediate effect on GEVCU operation. Once you have it trimmed up as you like, you must press **SAVE** button at screen bottom to send the data. These values are then copied into GEVCU EEPROM memory and retained until you change them again, either by serial port or wireless graphic interface.

Again, to save you will be asked for a password. It is **secret**.

There are a few special cases we should discuss here.

Almost all inverters monitor battery pack voltage and current and indeed report these values via CAN. So it is relatively trivial to calculate the use of energy over time in the form of kilo-Watt hours. Your battery pack has a certain amount of energy in it and you can calculate how much by multiplying the average pack voltage by the amp-hour rating on the cells. For LiFePo4 cells, the average or nominal voltage is typically 3.2v. So for a 100 cell 60 amp hour pack, you can assume about 320 x 60 kWh or 19,200 Watt hours or 19.2 kiloWatt Hours.

While you may be accustomed to counting energy in ampere-hours, kilowatt hours are slightly more useful. As your pack discharges, the voltage decreases. To put out the same power to the motor, it will have to deliver MORE current. And so this ratio constantly changes as the pack voltage decreases.

Measuring power in kilowatts basically factors this in. Voltage x current = power. And so 100 volts at 10 amps is 1000 watts but 50 volts at 20 amps is also 1000 watts.

GEVCU calculates the running total of kWh consumed in operation and displays it on a dial in the DASHBOARD tab of the web site. But there are a couple of provisos.

It only calculates it while in operation. It will use both positive currents and negative or regenerative currents to do this and it takes the measurement in very small time increments so it is reasonably accurate. But it does not monitor charging as the GEVCU and inverter are normally off for that procedure.

It WILL retain the value from one startup to the next and so it is cumulative as you use the vehicle throughout the day. But there is no real way to zero it on charge.

These types of systems tend to rely on the driver to "reset" the system after charging and personal experience would indicate that is a poor strategy.

So GEVCU automatically resets the meter based on the measured pack voltage. When you charge your pack completely, after charge the voltage will settle to some value we will call the FULLY CHARGED voltage. This should not be confused with the CHARGING voltage.

For example, lets say you had 100 LiFePo4 cells that you want to CHARGE to 3.65 volts each. You would set your charger for 365volts. It would charge the pack until it reached 365 volts, and then hold it there in constant voltage phase until the charge current diminished to some low value – 5 amps perhaps – and then terminates.

Within a hour or so, the voltage of the pack will settle to the open circuit voltage of the pack - typically in this case 332-338 volts.

When you first start the car and drive away, that voltage plunges immediately and dramatically to some lower value – typically 325 volts within the first block of travel. Even if the voltage was still up at 350 volts because it was actively charging when you started, it will still plunge to 325 volts very quickly.

On the GEVCU configuration page, there is a configuration variable for your battery pack voltage. You want to enter some value lower than the open circuit voltage of your fully charged pack there. Let's say 331 volts.

Any time and EVERY time GEVCU detects a pack voltage then higher than 331 volts, AND the current used is POSITIVE above zero, it will reset the meter.

This means that the meter is continuously being reset to zero until your pack voltage decreases below 331 volts. At that point, it begins to accumulate energy usage on the meter.

The requirement for positive current is to prevent it from resetting in the first few miles from regenerative (negative) currents that can cause voltage spiking on the pack briefly. In our experience, this voltage spiking is actually of no import and does no harm to the pack at all. To overcharge your pack you would have to come out of your driveway into a seven mile long downhill run. But we do not want to reset the meter on regen events.

You will find then that the meter simply reads zero until you've driven the car sufficiently to get the resting or static voltage below the value you entered, in this case 331 volts. With LiFePo4 cells, this is typically a BLOCK of driving distance. Subsequent to that, the meter will clock normally and it will save values to EEPROM so that it continues to accrue from one startup and drive to the next.

The one event it does NOT account for is partial charging. If you opportunity charge during your day, you will see that this has no effect on the meter unless you reach the fully charged state.

This is not a state of charge meter. It doesn't know how big your pack is, it simply shows you how much energy you have used. If you are aware that you have a 19.2 kWh pack and the meter shows you've used 17kWh, you can kind of know where you are at. If you want to limit your use to 80% depth of discharge, you would have to calculate that 19.2kWh x 80% equals about 15.4 kWh for example.

One final note. This meter only measures energy used by the inverter. It does NOT take into account the energy used by your air conditioner, your heater, your DC-DC converter, or anything else.

16 CANBUS COMMUNICATIONS AND OBDII

A central concept of the Generalized Vehicle Control Unit is that it interfaces with other automotive devices using the Controller Area Network or CAN protocol. CAN was originally proposed by Bosche in 1987 specifically for automotive applications.

More specifically, GEVCU is designed to control three-phase AC inverters – power switching electronics designed to drive AC induction and brushless DC motors – essentially all modern motors used in OEM electric car design. These “inverters”, including those from Toyota, Nissan, General Motors, UQM, Rinehart Motion Systems, Tritium, Tesla, and many others all use sensor management devices to gather driver inputs, and communicate that information to the “inverters” via CAN bus in the form of torque or speed commands to drive the motor.

The Atmel SAM3X8E ARM Cortex-M3 CPU used in the GEVCU actually has TWO independent CAN bus controllers built IN to the chip itself. But these controllers represent HALF of the necessary hardware to perform CAN communications. They also need CAN TRANSCEIVERS – devices that actually create the transmitted pulses on the bus.

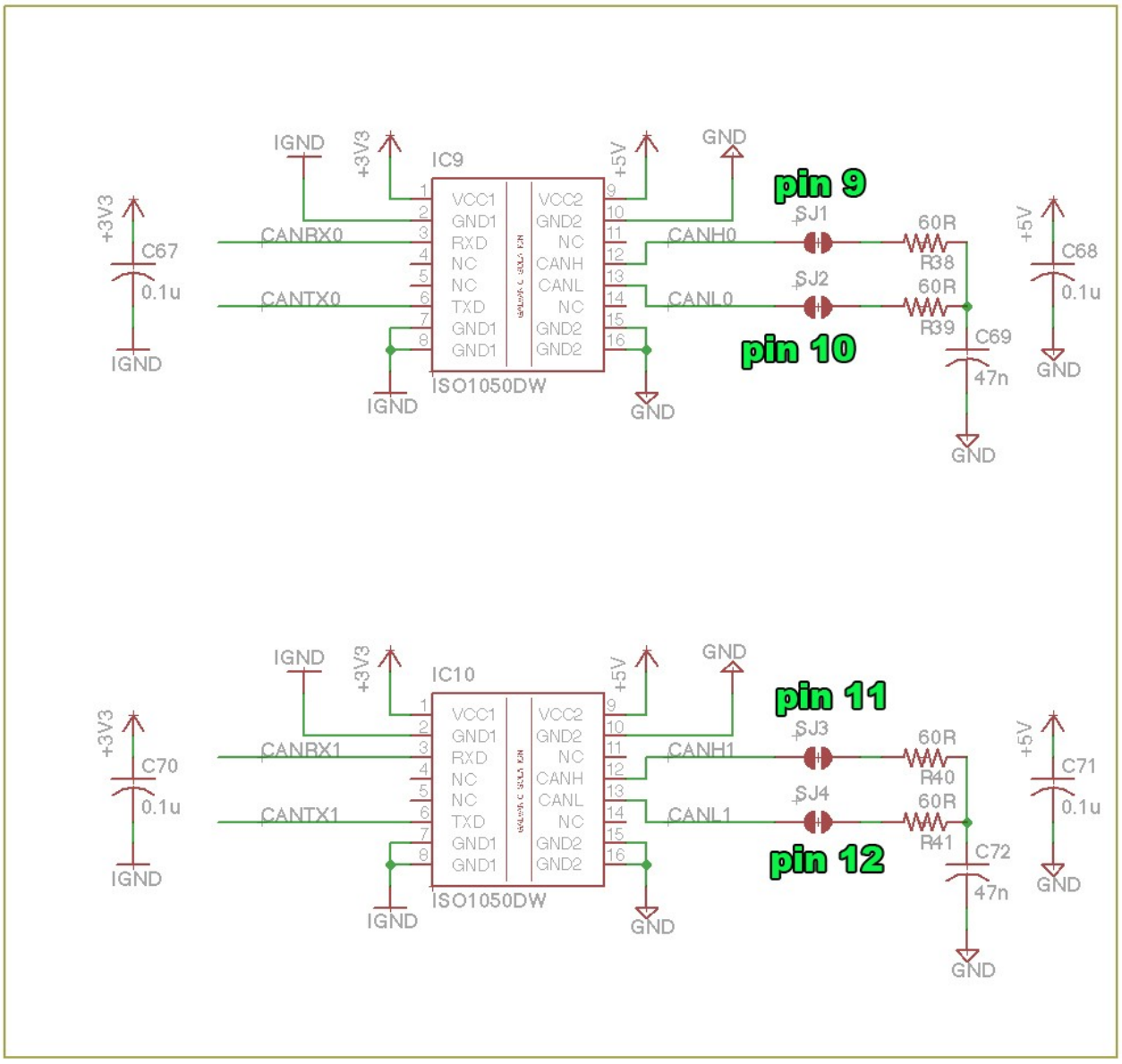
GEVCU uses two Texas Instruments ISO1050 galvanically isolated CAN transceivers that meet the specifications of the ISO11898-2 standard. The device has the logic input and output buffers separated by a silicon oxide (SiO2) insulation barrier that provides galvanic isolation of up to 5000 VRMS for ISO1050DW and 2500 VRMS for ISO1050DUB.

Used in conjunction with isolated power supplies, the device prevents noise currents on a data bus or other circuits from entering the local ground and interfering with or damaging sensitive circuitry.

As a CAN transceiver, the device provides differential transmit capability to the bus and differential receive capability to the Atmel CAN controller at signaling rates up to 1 megabit per second (Mbps). Designed for operation in especially harsh environments, the device features cross-wire, overvoltage and loss of

ground protection from –27 V to 40 V and over-temperature shut-down, as well as –12 V to 12 V common-mode range.

We refer to the two CAN buses as **CAN0** and **CAN1**. **CAN0** has a differential transceiver output available on pins 9 (CAN 0 high) and 10 (CAN0 lo) of the AMPSEAL 35 connector. **CAN1** similarly appears as CAN1 high on pin 11 and CAN1 lo on pin 12 of the AMPSEAL 35 connector.



CANBUS

Note the soldered jumpers **SJ1** through **SJ4**. These connect a 120 ohm terminating resistor on the CAN bus. To remove those, simply desolder the bridge to open it up.

For GEVCU end users, there is nothing you need to know about CAN. The wiring harness provided with the GEVCU already connects the GEVCU to the DMOC645 CAN bus, handles all terminating resistor issues, and has been tested repeatedly to

drive the DMOC645 and Siemens motor combination. There is really nothing to configure here.

But a bit of detail is provided here to illustrate the future growth and possibilities inherent in the GEVCU device.

The details of the CAN transmission are basically handled first at the transceiver level, and then at the CAN controller. But to effectively use all the mask and filter capabilities provided by these chips, and make CAN communications easier to program, a special software library is required to do CAN. This is the **DUE_CAN** library available at https://github.com/collin80/due_can. This allows fairly simple C++ commands to manage CAN communications quite effectively.

The ability to manage two CAN channels separately provides enormous flexibility. Of course, it could conceivably operate two separate inverters this way, each driving a different motor for example.

More commonly, it would be used as a CAN bridge, porting data from one CANbus to another. It is not uncommon to have CAN buses operating in the same vehicle at different speeds, one at 250 kbps and one at 500 kbps for example. Using GEVCU, it is relatively trivial to port data from one to the other.

The basic and immediate design of GEVCU is to command the Azure Dynamics DMOC645 controller directly via CAN. This is a 250kbps CAN bus that is simply two wires between GEVCU and the DMOC645.

The second bus would more likely be connected to the vehicles' **Onboard Diagnostics Version II** or **OBDII** bus. This has been required in virtually all vehicles worldwide since the late 1990s. Originally using a variety of manufacturer specific protocols, it has evolved in recent years ever more towards simply a J1939 standard CAN bus.

At the very basic level, the CAN protocol most often uses a 29 bit address and up to eight bytes of data. ANY device on the bus can transmit these packets and ALL devices on the bus can receive them. In broadest terms, the 29 bit address identifies WHICH device is sending the data, and often the specific nature or type of data it is sending. The eight data bytes then contain the data.

Using filters and masks, any specific device on the bus might set up to basically "look for" packets from another specific device, with specific data in it. It would ignore all other signals on the bus. And the software in the device would intelligently recognize not only the packet, and the eight databytes, but specifically the format and significance of every bit of the eight byte data payload.

Collision detection and priority are rather cunningly handled in the address itself.

And so we wind up with a bus, that might have two, but also might have two hundred devices, all more or less blindly sending packets and receiving packets. But by selectively filtering packets at the chip level, the receiving devices only get data they are interested in and know how to act on. The air conditioning controls on the dash have no use for inverter information generally speaking. The gear selector doesn't need any information at all, it simply transmits lever position periodically. The variable steering device only looks for RPM from the inverter. And so it goes.

Sending devices are more less simply "announcing" data that might be of interest to others by forming the data in agreed format, and sending it with the right identifying address – more or less one assigned to that device. Note the address is NOT who it is sending it to. It's just a message identifier showing the source device and nature of the message.

So GEVCU is completely capable of announcing drive commands from the address and in the format that is expected by the DMOC645. And it can also receive packets from the DMOC645 that contain data on inverter temperature, motor temperature, motor current, motor voltage, pack voltage, pack current, and much more.

Normally, GEVCU would get accelerator position from a hall effect pedal as an analog input. But it is not only completely possible, but has already been done, to connect the second bus to an OBDII bus and "capture" pedal position from the CAN bus signals transmitted by a CAN capable accelerator/throttle assembly. That data is then used to form drive commands going out the first CAN bus to the DMOC645.

It would be entirely feasible to program the GEVCU, using the PWM output of one of its digital outputs, to drive a vintage gas gage in a 1957 Porsche. But it is JUST as

feasible with GEVCU, to put the proper address and data format together on the OBDII bus to drive the gas gage in the instrument cluster of a 2014 Porsche.

This opens up enormous flexibility. For example, there are a number of devices on the market for trivial amounts of money ($20-40) that plug into the OBDII connector under the steering wheel of a modern car that transmit data from the CAN bus over either 802.11b/g Wifi or Bluetooth wireless. And there are already multiple versions of programs for the Apple iPhone, iPad, and Android tablets as well to capture CAN data from the vehicle and display that data on attractive and highly customizable gage displays. In this way, an Apple iPad could rather easily serve as a graphic display for your electric car, displaying kWh, amperes, battery state of charge, motor rpm and current, inverter temperature, and much much more.

The implications for CAN are actually hard to get your head around. For example, conventional thinking would indicate that GEVCU is quite limited with only 4 analog inputs and 4 digital inputs, and 8 outputs. Those accustomed to Arduino having over 50 inputs and outputs would find this very limiting.

In automotive applications, an evolution has occurred that is quite fascinating and perfectly logical. Look at the wiring harness that comes with the GEVCU. It is already sufficiently complex and enormous that we have individually colored each wire AND inscribed the logical label of the wire along the wire length – encased it in nylon braid, and it is STILL quite a thing to deal with in a car. It causes a lot of wiring.

The CAN philosophy is to reduce all that to two wires. If you need more than 4 analog inputs an 4 digital inputs and 8 digital outputs, you basically need ANOTHER CAN device closer to its logical purpose.

And so you would not incorporate measurement of battery voltage and temperature and current and state of charge in GEVCU. You would more likely devote an ENTIRELY SEPARATE device, located right AT the battery, and connect it to GEVCU with precisely TWO wires, the CAN bus wires. And so basically you have two wires running through the car, connecting dozens of highly specialized devices whose only wiring is very very short and very very local to just what it is doing.

Actually you COULD use two GEVCUs. One with the GEVCU software in it, and the other with battery monitoring software in it. The CAN bus is how they would communicate.

How far can this be taken? To the extreme. On modern cars, if you look at the 4 window controls on the drivers door, you will find it is a CAN device all by itself, and it communicates with three others – the window controls in the other three doors. The Chevy Volt actually has a total of **104 microcontrollers** in the vehicle, each with its own built in software and function. But they can all communicate with each other. The future of automotive technology looks like, sounds like you could check to see if any of your car windows were down, from your pocket phone, while in a building 112 miles away. Better yet, you could roll them up.

And so the real power of GEVCU, beyond giving us access to existing inverters and motors, is the two CAN bus channels. And added functionality is not so much a function of hanging more wires on GEVCU, but on intelligently designing and placing devices in the car to talk over the CAN bus. Fortunately, beyond perhaps a battery monitoring system, modern cars already have CAN for throttles, windows, door locks, radios, gas gauges, air conditioning, auxiliary fans, window washers, temperature sensors, and all of this is increasing at a logarithmic pace. If we can determine the address and commands, the open source nature of the GEVCU software will allow us to command it. Bosche actually invented CAN to REDUCE the weight and cost of copper in the wiring in an automobile. That was the original purpose of CAN. And it works.

Back to earth a bit, the GEVCU was born to drive the Azure Dynamics DMOC645 inverter and paired Siemens motor. But from the first instant of conception, we envisioned a very modular object oriented design leveraging the power of the C++ object-oriented language.

As such, a class **motorController**, is available. An object, inheriting from that class, is the **dmocMotorController**. It is actually a very small bit of program that intelligently takes concepts such as forward torque and regenerative torque, voltage, current, and temperature, and keys that to the SPECIFIC CAN messages and addresses EXPECTED by the DMOC645 already. And it intelligently knows how to recognize messages from the DMOC645, by address, and how to decode them to get actual torque, actual rpm, inverter temperature, battery pack voltage, etc.

To customize GEVCU to work with an inverter from UQM or Rinehart or Nissan Leaf, if you have the documentation defining the addresses and data formats used for those devices, it is a very doable if non-trivial programming task to add an object, much like **dmocMotorController**, to the **motorController** class. Ideally it's a single file. Call it **uqmMotorController**. The SPECIFICS for any controller are confined to really a tiny part of the GEVCU software. You don't need to know very much about how the rest of the program works, or why, to do this.

In the case of a UQM or Rinehart, these CAN data digests are actually published documents. In the case of the Nissan Leaf, it might be a little more difficult requiring some reverse engineering – basically driving a LEAF and sniffing out CAN codes on the CAN bus to the inverter. That's how we did it for the Azure Dynamics DMOC645.

But in this way, we hope to open the door to cogently reusing the cornucopia of excellent components deriving from the salvage of many many cars developed by the OEM manufacturers. Electric motors and controllers can conceivably operate for DECADES of use. But one tree planted in just the right place 30 years ago can take out a Nissan Leaf in a split second. The Tesla Model S is such a delight to drive, and will indeed do 0-60 in a little over 4 seconds. In fact it will do 0-45 and 45 BACK to zero in less than 4 seconds if it hits the right tree. And the Model S owners are out there desperately trying to prove us right on this theory.

And so we see a future land strewn ocean to ocean with glittering motors and inverters and battery packs that are virtually brand new, lacking only a car – and a device to talk to them nicely in a language they understand – GEVCU CAN.