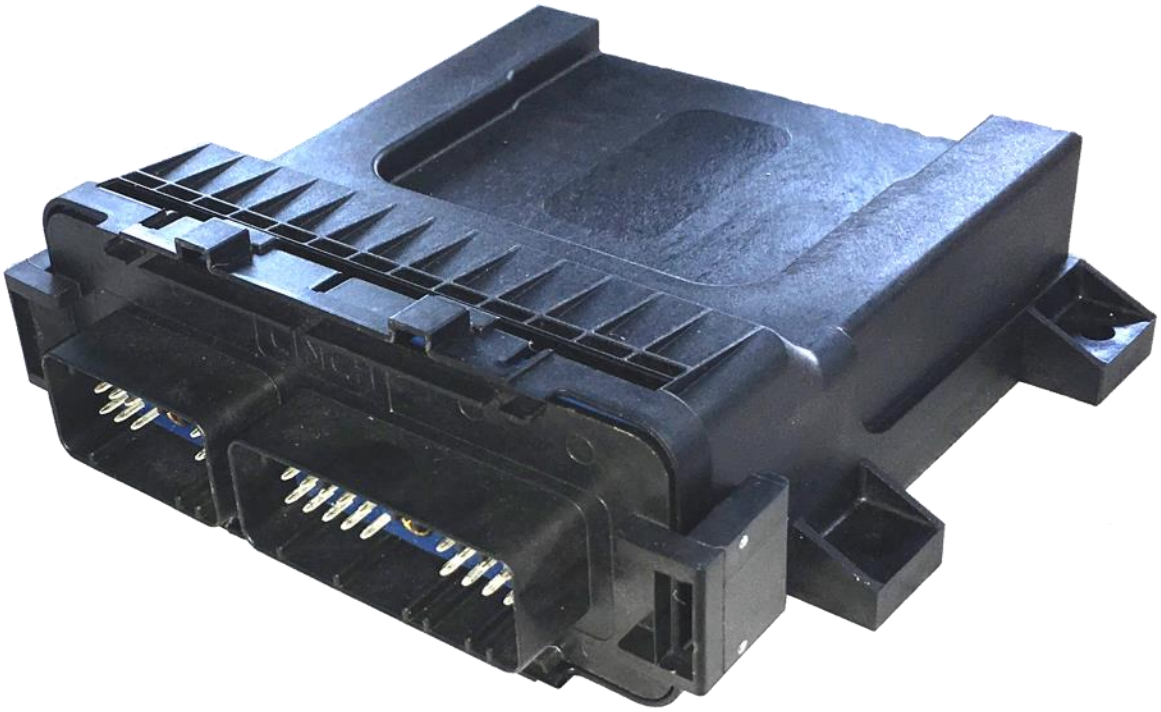
Generalized Electric Vehicle Control Unit

GEVCU

Version 7.00



LEGAL DISCLAIMER

This manual describes a hardware device produced by EVTV Motor Werks LLC. The **G**eneralized **E**lectric **V**ehicle **C**ontrol **U**nit or **GEVCU** is an experimental educational device designed to allow students and enthusiasts to explore and learn about electric vehicle control issues in automotive development – particularly using drive train components used by automotive manufacturers.

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 available at http://github/collin80/GEVCU7. 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 embodies 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, how well or poorly it does it, 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 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.

This document is intended to generally represent the GEVCU hardware sold and distributed by EVTV specifically. It reflects the software installed on the GEVCU hardware at the time of shipment. Obviously, if other software or modified software is loaded onto the GEVCU hardware, or indeed if the software is modified by the end user to provide other functions, the printed document and the device and its software would naturally be in conflict. This conflict could potentially pose certain safety issues to the end user.

It is both foreseeable and intended that other entities will also produce alternate GEVCU hardware and other forks or versions of the GEVCU software. Indeed this is already the case. Those entities should produce their own original documentation illustrating how THEIR version actually works at the time they shipped it. And they are specifically precluded by statute from distributing THIS document or any part of it with alternate hardware and/or software designs.

Table of Contents

1. **Introduction** 4
2. **Specifications** 9
3. **Wiring and Connections** 12
4. **Serial Port Interface** 15
5. **Module Selection** 18
6. **Precharge Considerations** 20
7. **Throttle Calibration and Mapping** 25
8. **Brake Calibration and Mapping** 31
9. **Power Values** 35
10. **Analog Inputs** 37
11. **Digital Inputs** 37
12. **Digital and Analog Outputs** 39
13. **Cooling Control – A Digital Output Example** 40
14. **Wireless Configuration**
15. **CAN bus Communications and OBDII** 42
16. **CAN Control of Input and Outputs** 46
17. **Updating GEVCU Software** 49

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 require 3 phase "inverters" to convert the DC power of the battery pack into three-phase AC 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 Bosch 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 computer or "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 hard-coded 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/software combination 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 vehicle 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, adapt, and extend 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.

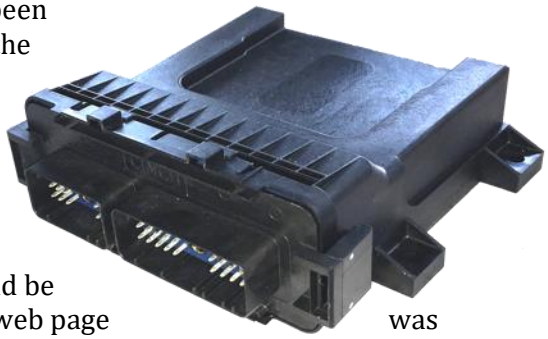
The operation of this device can be modified, within fairly narrow constraints, by a simple configuration "menu" style input that non-programmers can access and make changes to, in order to interface the VCU to the particular car they want to convert.

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.

GEVCU VERSION 6.22



While numerous software upgrades to the original GEVCU have been released under EVTV, and many more by other programmers of the device, the basic hardware of GEVCU has served remarkably well for over three years.

In 2016 Collin Kidder and Jack Rickard of EVTV embarked on a redesign effort for the device. Assembly of the original device was detailed and costly. While quite resistant to the rigors of the automative environment and weather, a full IP67 enclosure would be better. The Israeli wifi board used to produce the configuration web page was costly, hard to source, had a high failure rate and was difficult to configure. More importantly, other and improved interface devices had emerged.

And so GEVCU version 6.2 was developed. Improvements include:

1. Modice CINCH enclosure for IP67 environmental resistance.
2. Improved Analog to Digital Conversion inputs using 24-bit SPI converters.
3. Addition of isolated high voltage battery pack voltage measurement inputs.
4. A shunt input to allow measurement of high voltage system currents.
5. Blutooth 4.0 BLE data communications for iOS iPhone and iTablet and Android connectivity using the AdaFruit Bluefruit LE SPI module.
6. Optional GSM cellular data communications module using the Particle Electron.
7. Improved digital output control.

There were of course some casualties as well. The original concept of a built in web server for configuration was one of the central tenets of the device. No specific configuration program was required – any browser on any device could be used to configure it.

The USB port was exposed on the back of the unit. This was the main issue making the device vulnerable to weather and moisture. But it was very handy to plug into with a laptop to update software. Some users even extended this with a USB cable to offer USB in the dashboard or rear bumper of their vehicles.

The nature of the Modice CINCH enclosure makes this undoable. You must remove the board from the enclosure to access the USB port to update software.

GEVCU VERSION 7.00

The GEVCU 6.22 design was produced for a number of years but some shortcomings were to be found. Additionally, time marched on and hardware improved. The new GEVCU7 is a nearly complete redesign of the GEVCU6.22 to improve it in many ways.

1. The processor has been upgraded to a 600MHz Cortex M7 processor. This processor has been tested to be around 24 times faster than the processor in GEVCU6. It is much improved in storage space, RAM, and capabilities.

2. The processor and an add-on slot both use the MicroMod connector system from SparkFun. This allows for painlessly replacing the main processor should anything happen to it. It also yields a potential upgrade path should newer processors be available in the future

3. GEVCU7 now has 3 CAN buses instead of 2. Additionally, it can do CAN-FD on one of the CAN buses.

4. An ESP32 module is included on the board which allows for both WiFi and Bluetooth connectivity without needing to source an additional board as was the case with the GEVCU6 designs.

5. The inputs have been expanded. Now there are now 12 digital inputs and 8 analog inputs. There are still 8 digital outputs.

6. A microSD card slot has been installed on the board. This is most useful in order to be able to log the system performance while using GEVCU. The microSD slot can also be used to apply firmware updates to both the main processor and the ESP32 wireless connectivity.

2. Specifications


GEVCU-2c
EXT
www.EVtv.me

GEVCU PIN DEFINITIONS

18 PIN CONNECTOR

	1	2	3
A	VIN (+12V)	DOUT1	DOUT7
B	GND	DOUT0	DOUT6
C	GND	DIN10	DOUT5
D	GND	DIN11	DOUT4
E	EXT2	EXT1	DOUT3
F	EXT4	EXT3	DOUT2

30 PIN CONNECTOR

	1	2	3
A	AIN2	AIN1	AIN0
B	AIN5	AIN4	AIN3
C	AIN7	AIN6	A-GND
D	+5V	A-GND	A-GND
E	+5V	+3.3V	DIN7
F	CAN1-L	CAN1-H	DIN4
G	CANFD-H	CAN0-H	DIN5
H	CANFD-L	CAN0-L	DIN6
J	DIN9	DIN1	DIN3
K	DIN8	DIN0	DIN2

CAN1 is isolated and should be used for HV components

Table 2. Absolute Maximum Rating

Parameter	Symbol	Value	Units
Supply Voltage	VIN(+12V)	16	V
Regulated +3.3V output	+3.3V	400	mA
Regulated +5V output	+5V	700*	mA
Digital Outputs	DOUT0… DOUT7**	1.7	A
CAN BUS	CAN0 H/L; CAN1 H/L; CAN2 H/L	–27 to 40	V
Analog Inputs	AIN0… AIN7	5	V
Digital Inputs	DIN0… DIN7 DIN8… DIN11	20 10 !	V

*Total value for all pins
**Applying any voltage directly to any of these PINs will cause permanent damage to the GEVCU
! These digital inputs are meant to work as low as 5V for proper triggering. As such, they do not support quite as high of a voltage as the first 6 inputs. They will operate at 12V inputs but doing so may lead to overheating if input is high for an extended period of time.

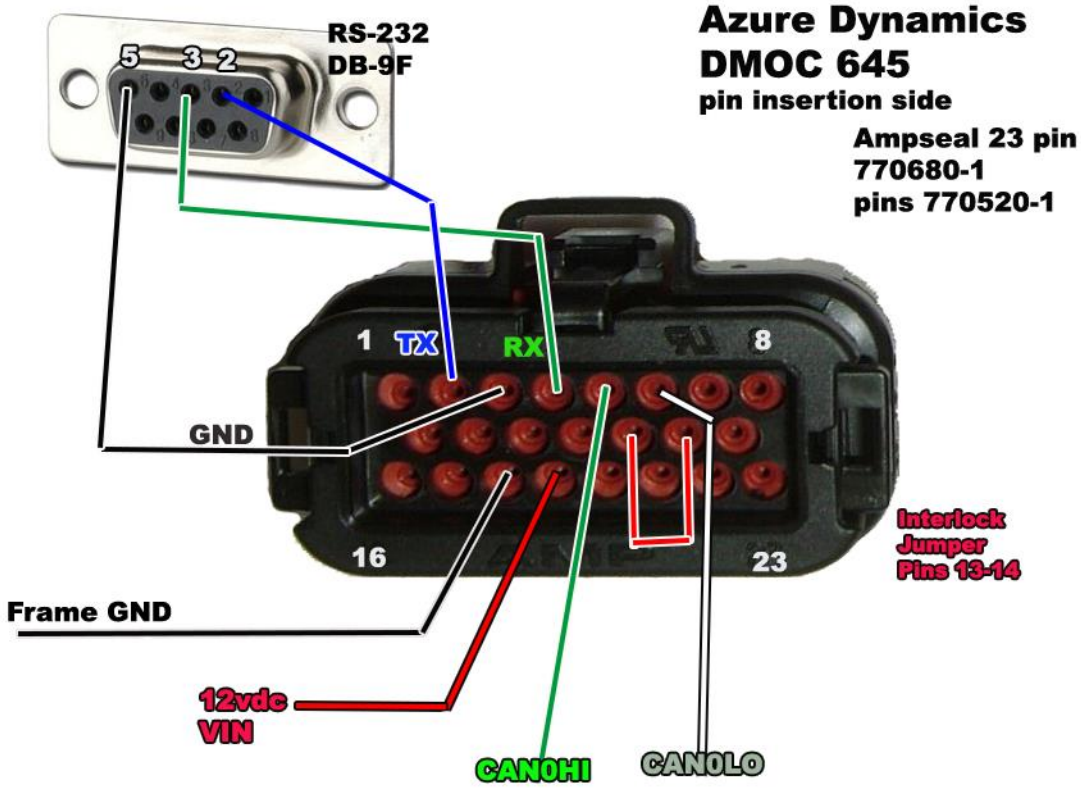
MICROCONTROLLER

Freescale IMXRT Cortex-M7
32-bit core
CPU Clock at 600Mhz.
1024 KB of SRAM.
16 MB of Flash memory for code
256KB EEPROM for persistent data.
Operating Voltage: 3.3v
Input voltage: 6-16v
CAN network channels: 3
Universal Serial Bus Port: 1
Analog Inputs: 8
Isolated Digital Inputs: 12
Digital Outputs: 8
Programming Environment: TeensyDuino 1.56

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 two Modice CINCH connectors.

Here is an example of the DMOC645 connector and which wires would need to be used:



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.

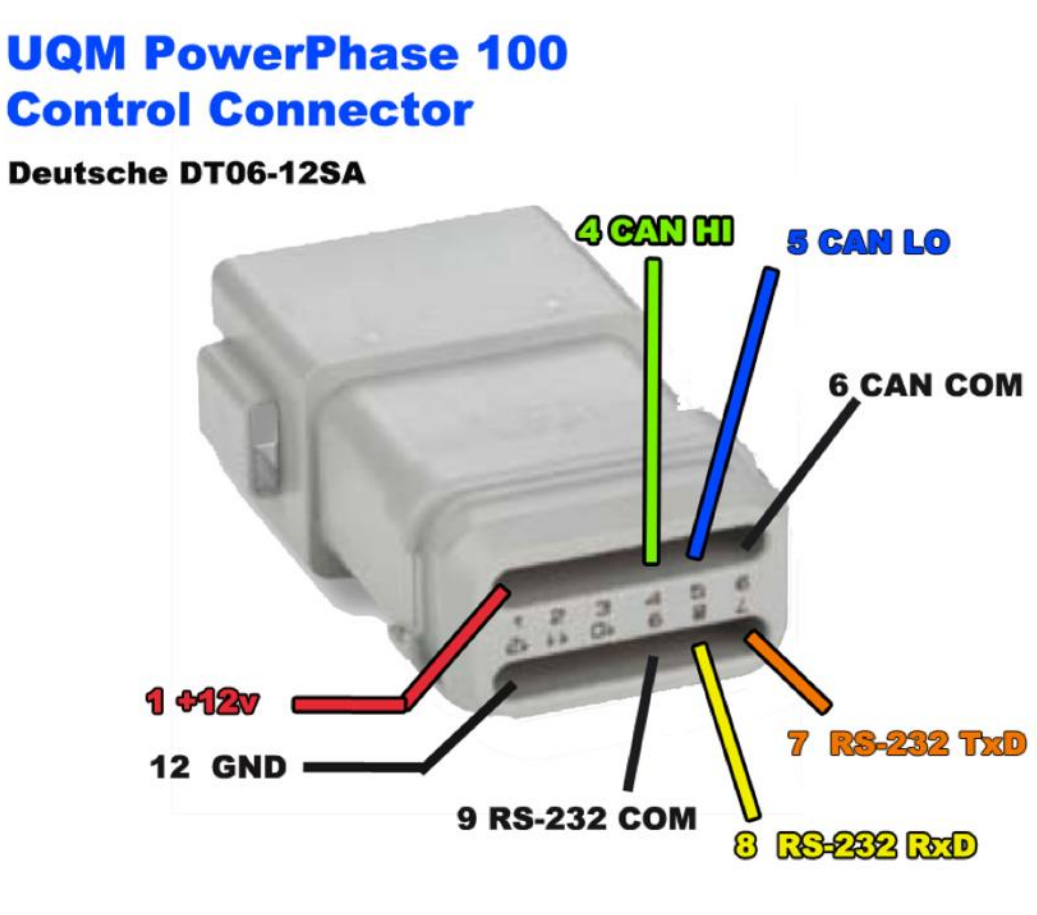
The two ModIce CINCH connectors provide but a subset of that but features inputs for 12v vehicle power, two CAN lines to the DMOC645 AMPSEAL 23-pin connector, eight analog inputs, twelve digital inputs, up to eight digital outputs and 3.3v and 5v outputs to power sensors.

GEVCU was originally devised for the Azure Dynamics DMOC645 Inverter and the basic wiring to this inverter is shown in the accompanying diagram. This is typical of the wiring necessary to connect the GEVCU to an inverter/controller.

Two connectors are provided with GEVCU with 18-inch wires already inserted. You may need to fabricate your own for your installation. Use the following Modice part numbers:

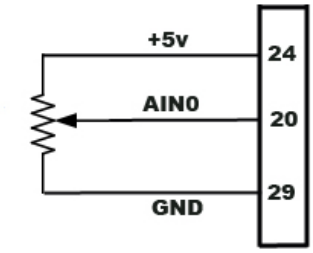
1. **18-pin connector: 538-581-01-18-023**
2. **30-pin connector: 538-581-01-30-029**
3. **18-gauge terminal pins 538-425-00-00-873**

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 typically rather dedicated to the motor inverter 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.

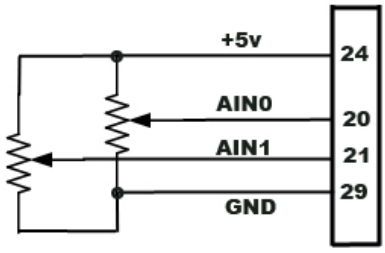
A basic throttle potentiometer is shown in the diagram . The potentiometer is a variable resistor that basically "divides" a voltage based on varying the point along a linear resistance where the voltage is sampled. This signal output is tied to our 1st analog input line AIN0 at pin A1 of the 30-pin CINCH connector. The voltage to be divided is provided by tying one end of the pot to our +5v output available on several pins for convenience – on the 18 pin connector 5v is available on A1/A2 and B1/B2/B3. The other end of the pot is tied to our reference ground at any row 3 pin of the 30-pin connector.

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 CINCH connector,

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.

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. Mobile tablet or phone interface via Bluetooth BLE
3. WiFi

In this way, the USB serial interface can allow us to avoid depending on specific operating systems or programs beyond a basic serial terminal program 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.

GEVCU features a printer style USB port on the rear of the circuit board simply because these appear to be the most physically durable and robust connectors for USB.

On power up, 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.

In practice, it is quite likely that your operating system will completely ignore all of the above settings as GEVCU7 presents itself as a native USB device and will send and receive at essentially full USB2 speed no matter what data rate you set. However, it will not hurt to use the above settings for 100% assurance that everything is set to reasonable values. 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.

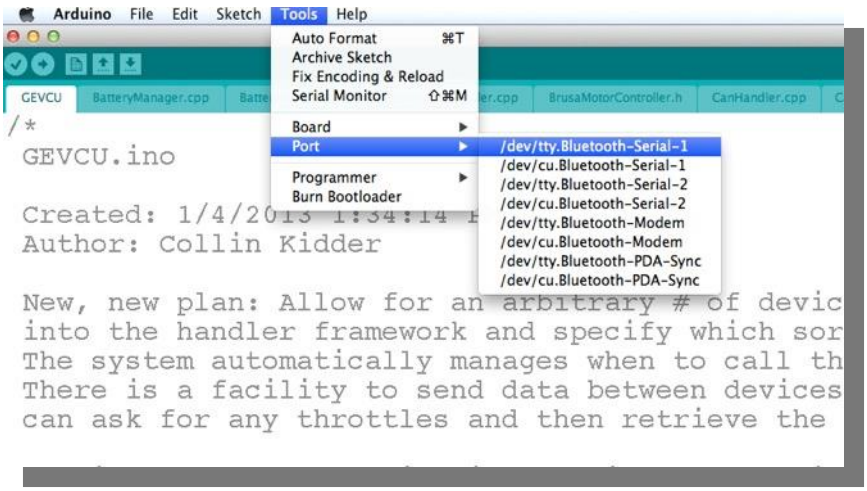
The Arduino IDE can be installed from arduino.cc
You will also need the TeensyDuino extension from PJRC

After installing both you can start the Arduino IDE. At the top of screen you will find menus. Select TOOLS, and in the submenu BOARD. Then select "Teensyduino" Finally, select "Teensy MicroMod".

Next, select the PORT submenu and 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 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.

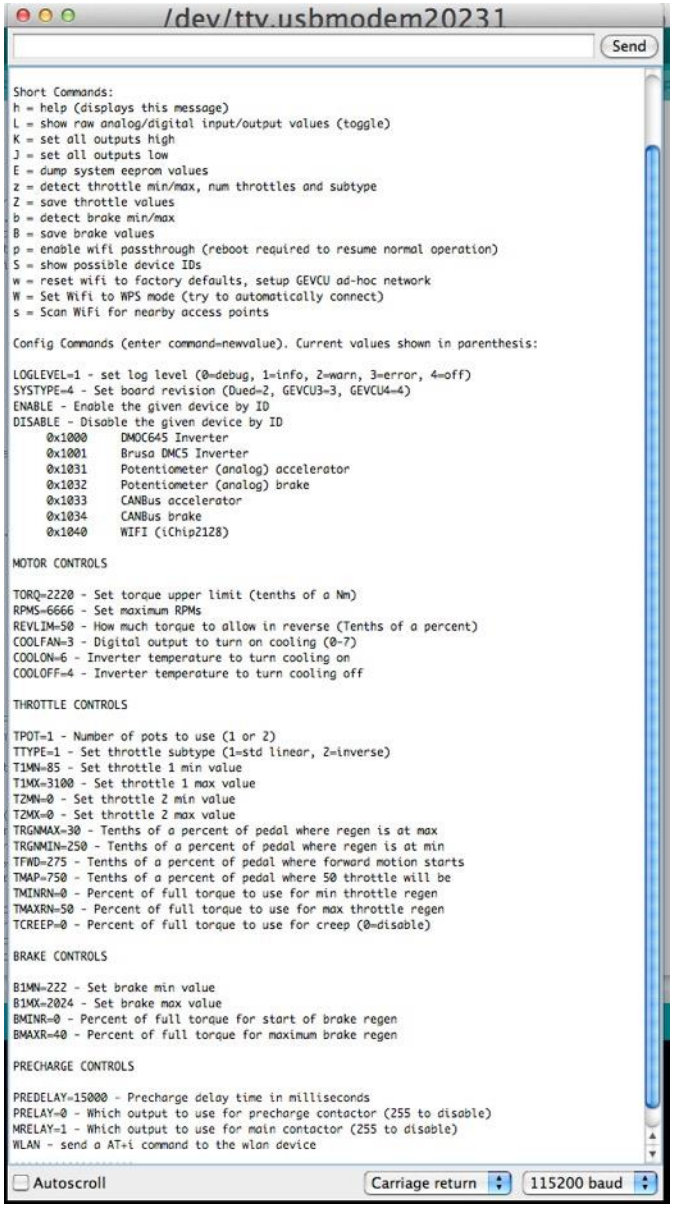
The only thing you will see initially is most likely a period that appears on screen. Every few seconds, you will see another one. This is the GEVCU heartbeat.

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.

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.

.



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 Digital Motor Controller (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 platform 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 UQM Powerphase 100 inverter. 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, chargers, instrument clusters, 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 CINCH connectors 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.

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

LOGLEVEL=1 Sets the onscreen reporting level of messages. The lower the level, the more verbose your output. Level 4 turns off all messages (not recommended). Level 3 adds error messages, 2 adds warnings, 1 adds informational messages. 0 turns on debugging messages which ordinarily will be way too much information but may be useful when attempting to determine why things are not working. Level -1 exists but shouldn't be used except during software development.

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

There are many modules already available for GEVCU7. You are also free to build your own. Further in this manual we will cover how to go about that. The system is now constructed such that the core has no hard ties to the modules and you can freely mix and match, adding and subtracting modules as needed. The system will automatically determine which modules exist and allow them to be used. As such, we cannot tell you in this manual which modules might exist at the time of your reading. However, you can get this information by sending the following command:

S Provide a list of all possible device IDs

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 (over 1000A is not impossible) 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 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.

The nasty math:

As a rule of thumb, one should pre-charge the capacitors for 3RC seconds. What is 3RC you might ask? The R in this equation is the resistance of the precharge resistor you are using. The C is the capacitance of the capacitors in your inverter (or other devices you need to pre-charge). You simply multiply them together then multiply by three. It should be noted that RC is the time it takes for the capacitor to 67% charge from the previous value. So, 1RC gets you to 67% charged. 2RC goes 67% of the remainder or 89% charged. 3RC gets you to 96% charged. In principle, this is sufficiently close to 100% to call it "good enough."

For instance, if you have 10,000uF (that's 0.01 farads) of capacitors and your precharge resistor is 400 ohms then the answer is 3 * 0.01 * 400 = 12 seconds. It should be obvious that we'd rather not have to pre-charge for so long. If the inverter really does have 10,000uF capacitance then perhaps we'd want a 100 ohm resistor instead. Now we can precharge in 3 seconds. Ultimately, the choice is yours. Smaller resistance values yield faster precharge times but require larger wattage resistors to compensate.

This brings us to the topic of resistor heating. The heating due to a resistance is I^2R where I is the current flow in amps and R is the resistance. So, if we have 4 amps at 100 ohms resistance we have 1600 watts of heating. However, life is not that simple. As the capacitor charges it increases in voltage. This brings it closer and closer to the pack voltage. As the voltage between the pack and the capacitor gets lower so does the amperage and thus the heating. At 200V difference you have 2^2*100 = 400 watts of heating. Much more manageable! In fact, most of the heating happens at the very start and rapidly tapers off. Even so, we are not discussing a situation where 1 watt resistors are sufficient. You will need a "surge" rated power resistor and it will be large. Expect the resistor to be at least the size of a pack of gum in order to have enough mass to absorb the full precharge. You might find resistors which are rated at 40 watts but 400 watts for a surge. If you get the proper resistance value this may work. Make sure to look at the specifications for the resistor and choose one that can handle the heating it will experience. When in doubt, get a bigger resistor. You won't hurt anything by overdoing the resistor size but you could start a fire or burn up the resistor by getting one too small.

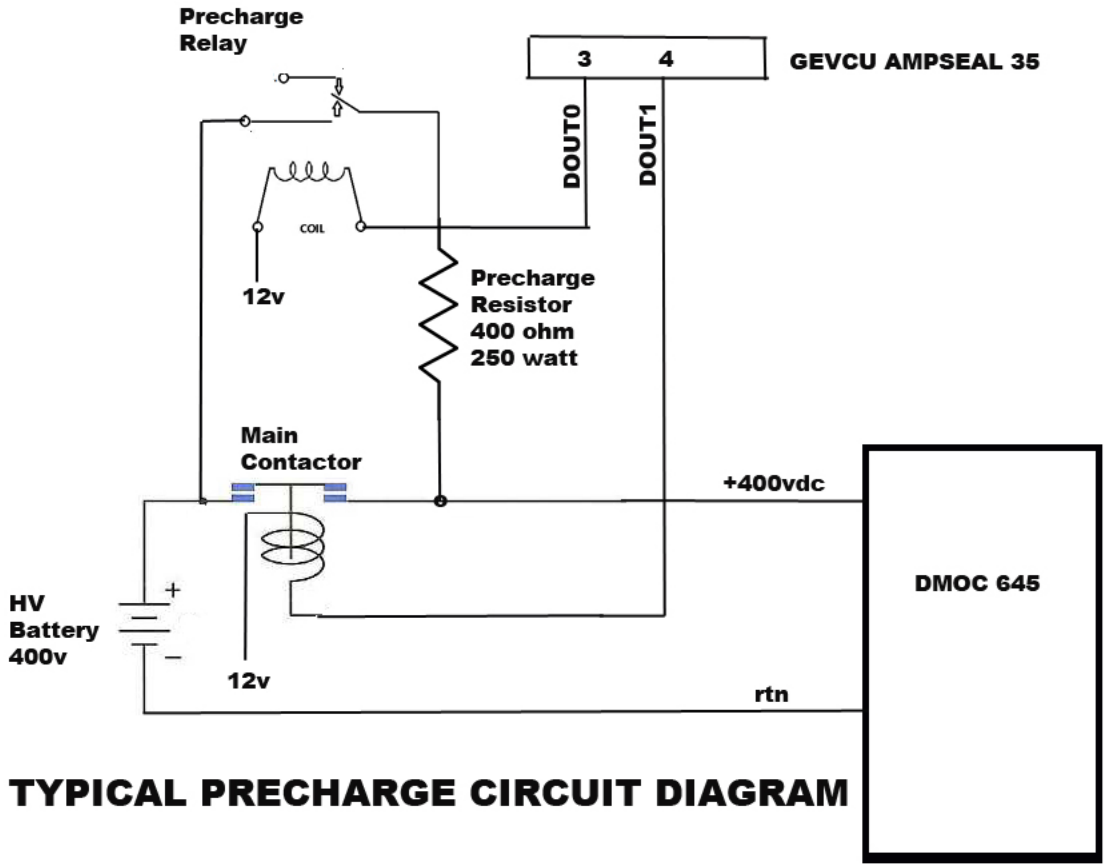
Unfortunately, once the caps are charged and the power switching begins, we don't want ANY resistance in our source voltage. As the switches often do 300 or 500 or even 1000 amps, it would give off a lot of heat and ultimately burn up even the largest resistor.

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 (or gets really close.)
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. However, since not everyone will want precharging, this functionality is not enabled by default. To enable it type:

ENABLE=0x3100 This enables the precharge module

Power cycle to enable the module

To set the precharge configuation, bring up the serial terminal on the GEVCU and enter the following values:

PRECHARGETIME=1500 This sets the precharge interval in milliseconds
Setting Precharge Delay to 1500 milliseconds.

PRECHARGERELAY=0 This sets the precharge relay output to DOUT0.
Setting Precharge Relay to 0.

MAINCONTACTOR=1 This sets the main contactor relay to DOUT1.

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 inverter.

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

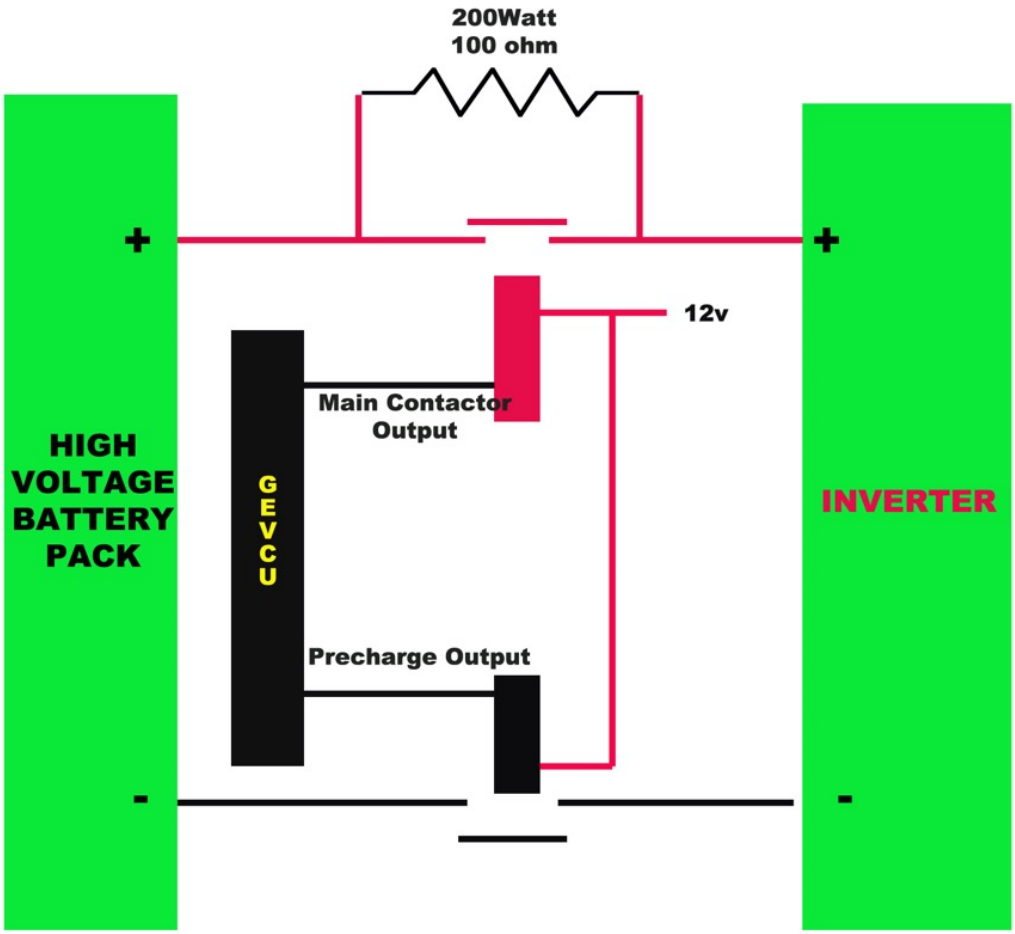
After that time has expired, it will close the **MAINCONTACTOR** output – in this case DOUT1 on pin 4.

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 **MAINCONTACTOR** and **PRECHARGEDELAY** to 255. You can then use those outputs for other purposes. Note also that no precharge is done unless a **PRECHARGETIME** 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. GEVCU will 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.

One last thing should be noted. This scheme protects the contactors and the rest of the system from undue stress upon start up. However, it does nothing to protect things at shut off. If at all possible you should NEVER turn off the GEVCU while the vehicle is in motion. Sometimes emergencies happen and you simply cannot help it. If life and liberty are in danger and the only solution is to turn everything off, by all means do so. However, opening contactors while large currents are flowing is likely to both damage the contactors and potentially zap your other high voltage items. Allowing GEVCU to power down and/or opening the contactors under load should only be done as the last resort.

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 by varying the excitation current in the motor stator windings.

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 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 Digital 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, and moon phases 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 0.83v and that is scaled and converted to a digital value of 680 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 AIN4: 81, AIN5: 81, AIN6: 87, AIN7: 77

DIN0: 0, DIN1: 0, DIN2: 0, DIN3: 0, DIN4: 0, DIN5: 0, DIN6: 0, DIN7: 0, DIN8: 0, DIN9: 0, DIN10: 0, DIN11: 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 eight analog inputs 0-7.

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 11 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. Actually you should enter a value slightly HIGHER than the value displayed to ensure that with no throttle input, we truly read 0 % throttle
7. Enter the second value noted using the command **T1MX=xx** where **xx** is the value displayed. And actually you would enter a value just a few points LOWER than what you read to ensure we have a 100% value at full throttle.

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

GEVCU also has an automated system that might be easiest. The "z" command will start automatic throttle calibration. It will ask you to leave the pedal at its rest position and then press it fully. This should find your throttle and how it maps. If it has worked and you are satisfied that it is correct you can use the "Z" command to save the new settings.

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.

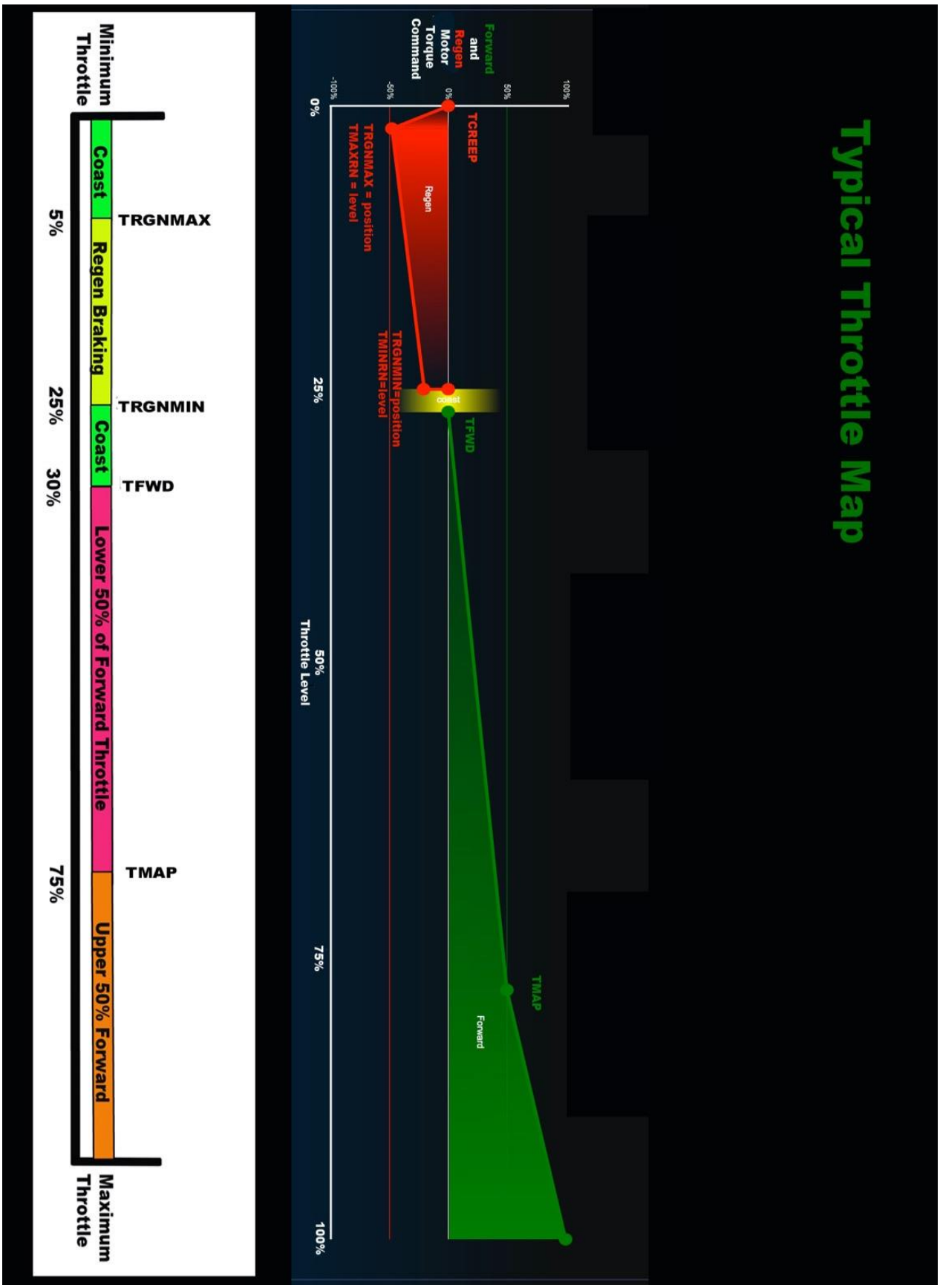
TRGNMIN – defines the point on the throttle where the minimum regenerative braking torque is applied. The minimum regen torque level is defined by an associated variable **TMINRN** which is the percentage of full torque.

TRGNMAX is the pedal position where maximum regenerative braking is felt and this maximum 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
Forward and Regen Motor Torque Command
100%
50%
0%
-50%
-100%
TCREEP
Regen
TRGNMAX = position
TMAXRN = level
TRGNMIN=position
TMINRN=level
coast
TFWD
TMAP
Forward
0%
25%
50%
75%
100%
Throttle Level
Minimum Throttle
Coast
Regen Braking
Coast
Lower 50% of Forward Throttle
Upper 50% Forward
Maximum Throttle
TRGNMAX
TRGNMIN
TFWD
TMAP
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.

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. Additionally, a non inconsequential amount of energy is spend in overcoming air resistance and tire resistance as you are driving. This energy is never coming back. 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 kinds of efficiency gain.

So why use regenerative braking at all? It actually puts more load and stress on the motor, the inverter, the transmission, 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. Also, it does save on brake pad and rotor costs.

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 Highway 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 of torque, the brake light output is set to ON.

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

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

Brake light output updated to: 5

REQLEVEL=-200 This sets the required regen torque to trigger the brake to be 20Nm (regen torque is negative and value is passed in 1/10th Nm increments)

Brake required torque updated to : -200

If you do not need a brake light output, set this value to **255**. Also note that the lighting control code may not be enabled. You can enable this by typing **ENABLE=0x3300**

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. You would, after all, probably find it unpleasant if your car tried to accelerate to 60MPH fast as physically possible. Instead, 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 high 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 command. 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 in watts. 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. Near the max RPM the available torque will be scaled back to prevent further acceleration.

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. This command may be handy as it is rarely wise to accelerate heavily in reverse.

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

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 analog inputs to the microprocessor.

GEVCU6 used an isolated 24-bit analog converter but this proved to be overly complicated and prone to excessive noise. GEVCU7 now uses filtering and buffering which as proven to provide very good analog performance with minimal parts. Additionally, this has allowed us to provide twice the number of analog inputs. It is now possible to use 8 analog inputs for various uses. They are, however, all 0-5V inputs not 12v.

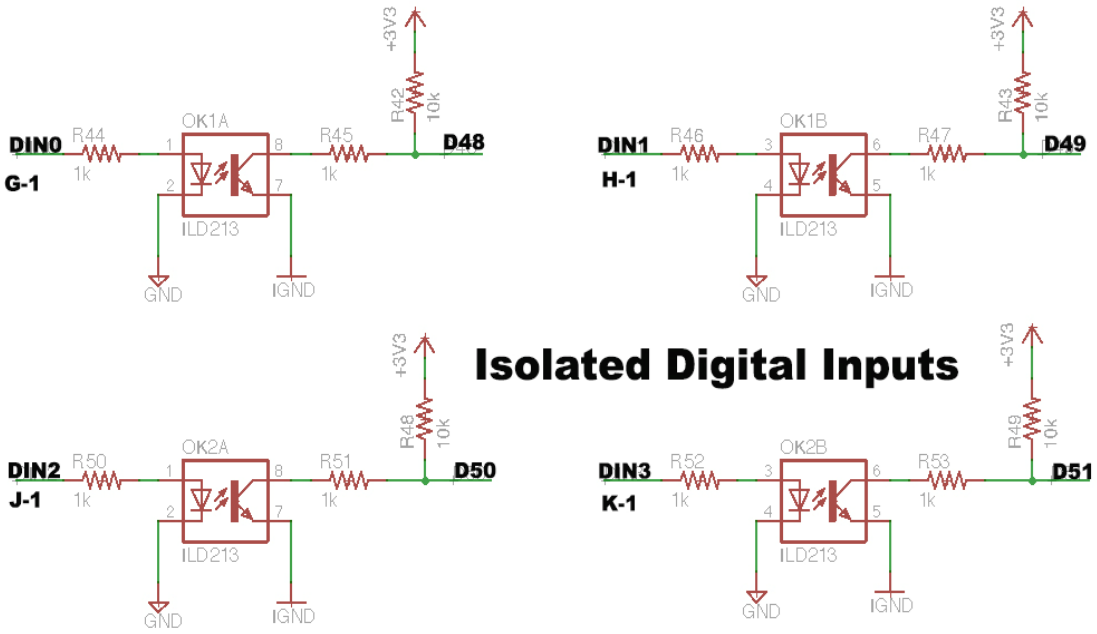
So far, by convention we have used inputs AIN0 and AIN1 for throttle, input AIN2 for brake and input AIN3 is available.

11. Digital Inputs

The GEVCU provides twelve 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 1k ohm current limiting resistor allows for an approximately 14ma input to the opto-isolators. This lights an LED which puts the output into conduction. In this way a 12V signal can trigger the digital input without having any direct connection to the processor. 8 of the 12 digital inputs have the 1k ohm resistor. The other 4 use 470 ohm resistors. Why the difference? 470 ohms is small enough that 5V will reliably trigger the inputs. The inputs with 470 ohm resistors can still be used with 12V for short durations. But, long term use of 12V signals to these digital inputs will likely burn up the current limiting resistors. As such, if at all possible it's best to use DIN0-D7 for anything that might have 14 or more volts.

The output uses a 10k pullup resistor to the isolated 3.3v supply of the 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.



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 **ENABLEIN** command.

ENABLEIN=2 will set digital input DIN2 as the ENABLE input. When 12v is present on CINCH pin J-1, 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 **ENABLEIN** has a value of **0-11**. If you do not need this feature, set **ENABLEIN=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 **REVIN** input.

REVIN=2 will set digital input DIN2 as the reverse input. When 12v is applied to CINCH pin J-1 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 **REVIN** is set to **0 - 11**. To disable the reverse function, set **REVIN=255.**

12. Digital Outputs

GEVCU provides 8 digital outputs any of which can also be used to switch devices on or off.

The basic 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.

Instead, the outputs from the microprocessor trigger MOSFETs which do all the heavy lifting. These MOSFETs can easily handle 2A continuously and much more as a surge. All outputs are protected by self-resetting fuses.

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 separate cooling loops using a Piersburg pump which does about 17 liters per minute through the AN-6 braided nylon hoses. 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.

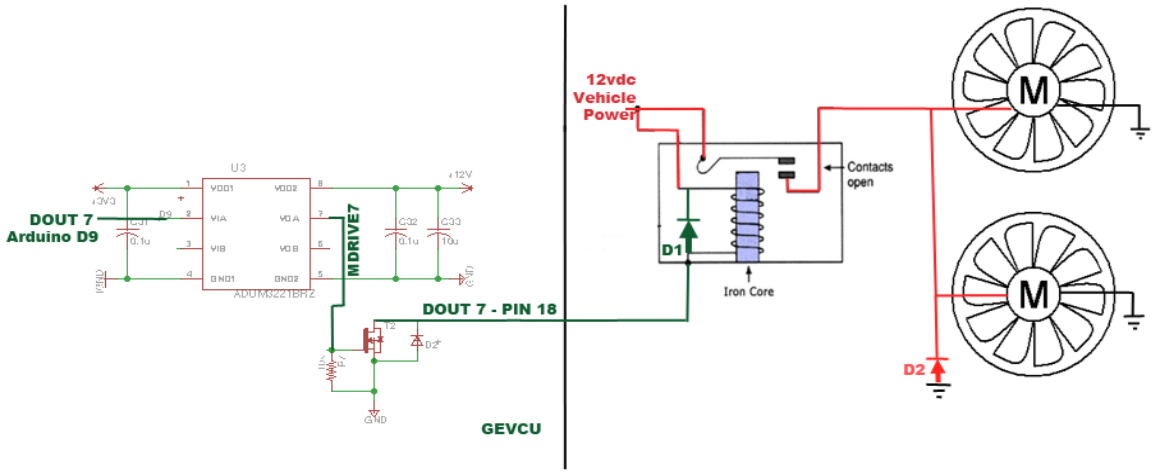
Of course, the fans not only use quite a bit of 12v energy, but they also are uncomfortably noisy. This is not so much a factor on the highway, but tooling around town it's a little loud.

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. This saves energy, and allows us a very quiet car when maneuvering at low speeds, while providing maximum cooling on the freeway.



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 F-3 on the CINCH 18-pin 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. Diode 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=45 This sets the temperature at which the output goes active.

Cooling fan ON temperature updated to:45

COOLOFF=35 This sets the usually lower temp at which the output becomes inactive..

Cooling fan OFF temperature updated to: 35

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. 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 microcontroller used in the GEVCU actually has THREE 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.

As a CAN transceiver, the device provides differential transmit capability to the bus and differential receive capability to the CAN controllers 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 three CAN buses as **CAN0, CAN1** and **CAN2**.

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.

The ability to manage three 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.

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 gauge 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 8 analog inputs and 12 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 8 analog inputs, 12 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. Actually that's how we did it for the UQM Powerphase 100 as well.

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.

16. CANBUS INPUT/OUTPUT CONTROL

As described earlier in this manual, GEVCU features 8 analog inputs, 12 digital inputs, and 8 digital outputs. Version 7 of the software provides for some CAN access to these inputs and outputs.

This can be very useful. Another CAN device on the bus might have access to information or sensor data not available on GEVCU, but be limited in input and output capability. Via CAN message, it could manipulate GEVCU inputs and outputs.

For example, we recently came across a very nice switch panel made by DNA termed the Powerkey Pro 2400. It features 8 configurable switches with LED backlights. The switch position is transmitted via CAN and the LED light state can be controlled by CAN.

Similarly, another device may want to know what the state of GEVCU digital input 3 is at any particular time. It can easily retrieve this information from the GEVCU.

CAUTION

This ability does engender some danger of conflict. If you use digital output 4 to close your contactor, and then use CAN to set output 4 to zero, it is hard to predict whether internal algorithms or the CAN command will win and for how long. So make sure to coordinate any use of input and output with your assignments of digital inputs and outputs in the normal GEVCU configurations.

GEVCU is configured to use four CAN bus message IDs to deal with inputs and outputs.

0x606 Received by GEVCU to set digital outputs 0 through 7
0x607 Transmitted by GEVCU to advise state of digital outputs 0 through 7.
0x608 Transmitted by GEVCU to advise state of analog inputs 0 through 7.
0x609 Transmitted by GEVCU to advise state of digital inputs 0 through 11.

GEVCU normally does NOT transmit **0x607** or **0x608**. It will ONLY produce these advisories if it receives a **0x606** message.
You may quite commonly wish to retrieve data on the state of GEVCU I/O without actually commanding a switch. You can easily do this by transmitting a **0x606** with all eight data bytes set to zero.

NOTE

The message IDS used by GEVCU for CAN I/O management are arbitrary. It will come with these set to these addresses. But this information is configurable. The file config.h contains defines for CAN_SWITCH, CAN_OUTPUTS, and CAN_INPUTS. You can change these values in the source code to whatever you like. But you must recompile and reupload the software for these changes to take effect.

GEVCU normally has two CAN busses configured and they maybe configured at different speeds. Understand that CAN IO commands can be sent and received on EITHER bus at any time. No configuration of this is required. Advisory responses will appear on the SAME bus the command came in on.

SETTING GEVCU DIGITAL OUTPUTS

Setting GEVCU digital outputs is limited to simply setting them to 1 (high) or 0 (low). No PWM capabilities are provided via CAN.

The message should in all cases be 8 bytes long with byte zero corresponding to GEVCU digital output 0 and byte 7 corresponding to GEVCU digital output 7.

There are three recognized commands for EACH digital output.

0x00	Make no changes. Do not disturb this output
0x88	Set this output to 1 (HIGH) no matter what its current state is.
0xFF	Set this output to 0 (LOW) no matter what its current state is.

By way of example, consider the following message:

```
0x606 00 00 FF 00 00 88 88 00
```

On receipt of this message, GEVCU would set digital output 2 to 0 and digital outputs 5 and 6 to high (1). Note that it would not disturb outputs 0, 1, 3, 4, and 7 at all.

READING GEVCU DIGITAL OUTPUTS

On receiving a **0x606** command, GEVCU not only sets the digital outputs, but also produces two messages – **0x607** advising GEVCU digital outputs and **0x608**, advising GEVCU inputs.

Message ID 0x607 is basically a modified echo of 0x606 showing the state of each of the eight outputs.

```
0x606 88 88 FF FF FF 88 88 FF
```

In this example, outputs 0, 1, 5 and 6 are all set to zero (low) while outputs 2, 3, 4 and 7 are set to high.

You can use this to minimize traffic on the CAN bus. For example, you might send the following message:

```
0x606 00 00 00 00 00 88 00 00
```

You can send this message periodically until you receive a message 0x607 showing output 5 as set.

```
0x607 88 88 FF FF FF 88 88 FF
```

At this point, we see that output 5 is 88 indicating a set condition. We really don't need to transmit anymore. And if we don't transmit, GEVCU doesn't reply either.

You could also use byte 5 of this message to turn on the backlight LED behind the switch, or another LED indicator showing the state of this output.

Anytime you subsequently just want to check the state of the outputs, or inputs for that matter – simply send a **0x606** with no changes

```
0x606 00 00 00 00 00 00 00 00
```

This will cause an automatic transmission of all advisory response messages

READING GEVCU ANALOG INPUTS

Anytime GEVCU receives a **0x606** message, it will also output a CAN message **0x608** containing the measured values on the four analog inputs to the GEVCU. These are two byte integers with the high byte first and the low byte second (MSB/LSB). To convert this to a readable number, multiply the high byte by decimal 256 or hexadecimal FF and add the low byte to the total.

All GEVCU analog inputs are read in 12-bit mode and so will appear as a digital value between 0 and **0x0FAC** (4012 decimal).

```
0x608 0F 21 0A F0 10 E9 22 C7
```

In this example, Analog input 0 appears as the first two bytes 0 and 1 as **0x0F21** or **3873**.

Note that you can freely read these instantaneous values at any time, even if they are used for example for throttle input without interference. But these are raw instantaneous values – not the highly averaged and smoothed values used for actual throttle control.

READING GEVCU DIGITAL INPUTS

Anytime GEVCU receives a **0x606** message, it will also output a four-byte CAN message **0x609** containing the measured values on the four digital inputs to the GEVCU.

This will use the same FF/88 signaling as our other digital outputs.

```
0x609 FF FF 88 FF
```

Here we see that digital inputs 0, 1, and 3 are all zero but digital input 2 is active (high).

17. UPDATING GEVCU SOFTWARE

It is entirely possible for you, the end user, to download the IDE, the extensions, all the libraries, and update GEVCU yourself. It's possible! But this may be beyond the ambition of many of the GEVCU users. And so we've developed a kind of shortcut method to update the software that many will find easier.

The Generalized Electric Vehicle Control Unit is an open source hardware and software device designed to empower builders and experiments in their efforts to build or convert one-off custom electric vehicles.

A version of the GEVCU hardware is produced and sold by Electric Vehicle Television.

The software of GEVCU is developed in and is fully compatible with the Arduino Integrated Design Environment (IDE). It uses a very C++ syntax with full object-oriented architecture.

The development of this software is ongoing and continuous with numerous contributors adding features, object modules and utility to the original software package.

EVTV maintains a github repo with the latest source code for GEVCU that is certified to work with the GEVCU hardware that EVTV provides. This can be found at http://github.com/collin80/GEVCU7. The Master branch is stable and should be used by people who are interested in having a known working system. Other branches may exist, WIP for instance. No guarantees are offered for suitability, operation, or functionality of these other branches. However, they may be more recent and have features that you are looking for. As always, sufficient testing should be done before using such code on a vehicle on the public roads.

Since the software changes on almost a continuous basis, you will from time to time desire to update the software held in flash memory on your GEVCU device.

For the programming savvy, this easily accomplished by downloading the desired version of GEVCU source code and compiling and uploading to the GEVCU device using the Arduino IDE.

But for many GEVCU users, C++ is a bridge too far and they want to be able to use the many features of GEVCU and indeed the latest version, without installing the Arduino IDE or doing any compiling.

For those, we have developed a very easy binary software update procedure that is very quick and basically involves a single click of the mouse to update a GEVCU connected to a personal computer via USB cable.

Our binary updater can update your GEVCU7 over USB from either LINUX or Windows. Simply download the latest GEVCU7-Updater.zip file, extract it, and run flash.bat from Windows to start the automatic process.

It is also possible to update both the Teensy MicroMod and the ESP32 from the sdCard. We will also provide binaries that can be extracted onto an sdCard and inserted into the slot on the GEVCU7 board. Simply power up the board thereafter and wait until it is done updating (much easier to see this from the USB based serial terminal). This method has the advantage of being able to easily update the ESP32 as well. The ESP32 contains an embedded webserver that hosts a webpage that can be used to monitor the status of GEVCU7 and configure it.