GEVCU

Generalized Electric Vehicle Control Unit

Programmer Checklist
For Adding Object Modules to the GEVCU Software

The Generalized Electric Vehicle Control Unit software is open source and object oriented. It was originally developed to control the Azure Dynamics model 645 Digital Motor Controller (DMOC) – a 3-phase inverter/controller used with the Siemens 1PV5134 motor on the Azure Dynamics eTransit Connect.

The DMOC645 was CAN controlled and did not feature any inputs for throttle, brake or other inputs and outputs. The eTransit Connect Vehicle Control Unit (VCU) performed that duty. Lacking the VCU, we had to develop a VCU from scratch capable of providing the necessary CAN messages to the DMOC645 to in turn control the motor.

The concept of a GENERALIZED VCU specifically for electric vehicles derived from the many different types of cars that would be converted to electric drive using this drive system. But from the beginning we saw the need to drive other types of controller/inverters and indeed interact with other devices such as chargers, DC-DC converters, and battery management systems.

So a central tenet of the GEVCU was a very object oriented approach that would easily allow the addition of new classes and new objects to accommodate different equipment. The goal of course to limit software changes to just the additional object added to accommodate the new equipment.

Some wins and some losses. Indeed GEVCU now supports the BRUSA drive train, the CODA/UQM drive train, and more are envisioned and already a series of chargers, BMS, and other devices have been added.

But along the way, we encountered several targets of opportunity that were just two good to pass on, that tend to complicate the easy addition of object modules.

CREATE OBJECT MODULE

TASKING AND PRIORITIES.

PERSISTENT CONFIGURATION STORAGE

MULTIPLE DEVICES ON CAN

CREATING AN OBJECT MODULE

You create an object module basically by adding a .cpp and .h file to the program set defining your object module. The best way is to copy one from a similar device. If you are doing an object module for a motor controller, a good start would be to copy and rename the DMOC645 motor controller object module held in `DmocMotorController.cpp` and `DmocMotorController.h`

Let's call the new module EVIC and we'll rename the copied files `EVIC.cpp` and `EVIC.h`.

To add these to the program, you basically do two things.

1. Add an include for `EVIC.h` in the `GEVCU.h` header file.
   ```
   #include "EVIC.h"
   ```

2. Add a createObjects entry in the main program file `GEVCU.ino`

   ```
   void createObjects() {
   PotThrottle *paccelerator = new PotThrottle();
   CanThrottle *caccelerator = new CanThrottle();
   PotBrake *pbrake = new PotBrake();
   CanBrake *cbrake = new CanBrake();
   DmocMotorController *dmotorController = new DmocMotorController();
   CodaMotorController *cmotorController = new CodaMotorController();
   DCDCController *dcdcController = new DCDCController();
   BrusaMotorController *bmotorController = new BrusaMotorController();
   ThinkBatteryManager *BMS = new ThinkBatteryManager();
   ELM327Emu *emu = new ELM327Emu();
   ICHIPWIFI *iChip = new ICHIPWIFI();
   EVIC *eVIC = new EVIC();}
   ```

Here, we instantiated the object eVIC of type EVIC as a new instance of EVIC().

Our eVIC object is now part of the program and enjoys all the rights and privileges of other objects of the program.

Note that most objects you will create will actually be child objects of more generalized parent objects. A motor controller object, for example, usually inherits from MotorController and from CanHandler. Much of the basic work common to all motor controllers is already done for you in the parent and standard variables are already there for you to update from your own specific code. You also generally must inherit from CanHandler to be able to catch and send CAN messages in your object.

So you will find something like this in your .h file.

```
class EVIC: public Device, CanObserver {

    public:
```

Here, in EVIC.h, we are defining a new class EVIC which inherits from the generic **Device** class but also from **CanObserver.** **A motor controller might be better to inherit from MotorController and CanObserver.**

Obviously, you will likely have to rename almost all the class references in your .cpp and .h files.

TASKING AND PRIORITIES

We hesitate to claim we have duplicated Linux and a real time operating system here. But the Arduino Due does sport an interesting set of clocks – 3 of them actually – each with 3 configurable interrupt timers for a total of 9 timer interrupts.

Inverter/controllers are operating powerful motors in real time and can be a bit demanding in that they want constant attention – regularly updated CAN control messages or they tend to shut down as a safety consideration. So inverters need the highest priority.

GEVCU does provide information to the user and allows configuration either by USB serial port or via a wireless web site. But while driving down the road, it would be quite normal for NO ONE to be using this at all to configure the system. So this is an example of a very low priority. Chargers and Battery Monitors and so forth would fall somewhere in between.

So we have developed a module called a TICK HANDLER. Your object module must register with this TICK HANDLER and then provide a public method, also termed TICKHANDLER to perform its duties as it can when the TICKHANDLER passes it a tick (timeslice).

You can think of your tickhandler as the LOOP routine in the Arduino Sketch/Processing environment or more traditionally MAIN.

To register your object with the tickhandler, the following general steps are in order.

1. Add a tickhandler DEFINE for your object to **`config.h`** file to set your tick interval in microseconds.

   ```
   #define CFG_TICK_INTERVAL_EVIC   100000
   ```

   Here, we define a tick interval for EVIC to call it every 100,000 microseconds (100 milliseconds)

2. Add a setup function to your object module establishing tickhandler

   ```
   void EVIC::setup()
   {
   TickHandler::getInstance()->detach(this);//Turn off tickhandler

     (Any number of other setup statements)

   TickHandler::getInstance()->attach(this, , CFG_TICK_INTERVAL_EVIC);
   }
   ```

In case our tickhandler is on, we turn it off. Perform any other setup functions desired. And then attach the tickhandler using the DEFINE from config.h.

3. Add a tickhandler function to your object module

```
void EVIC::handleTick()
{
      sendCmdCurtis();
      sendCmdOrion();
}
```

Here, we have a short function that calls two other functions. Each of those create and send a CAN command. So each 100 ms, the tickhandler will call the object function handleTick, which will send two CAN messages and return.

4. In your object module .h file, add the following includes:

```
#include "config.h"
#include "TickHandler.h"
```

PERSISTENT CONFIGURATION STORAGE

Almost all modules in the GEVCU have some level of need to store information between operating sessions. This may be system data that you want to maintain from startup to startup such as accumulated Ampere-Hours or peak temperatures.

But almost all devices have some configuration options that end users can adjust to modify the operation of the software.

The Arduino Due ARM3 processor gives us very good power and speed for the VCU function, but it does not provide an EEPROM and the use of flash memory

for persistent data has a problem in that it is totally erased if the software is updated.

So GEVCU has a 2Megabit (256kbytes) EEPROM provided just for such persistent data.

In coding independent objects that could be enabled in a variety of combinations, we have to have some means of rationalizing who stores what at what address and when. Otherwise, objects would be overwriting each other's data.

Further, EEPROM's have a defined write life. Not quite the issue it was originally, it is still nice to limit the number of writes so that the same data is not rewritten every 30 msec.

We do this with a device termed a MEMCACHE. An object has to register with DEVICES and is allocated a block of memory in MEMCACHE. The data in MEMCACHE is then recorded to EEPROM periodically. And it is loaded into MEMCACHE on startup. In this way, you can save your configuration data and load it again on the next startup.

1. Each device should create a configuration class that inherits from **DeviceConfiguration**. This will be in your .h file.

```
class EVICConfiguration: public DeviceConfiguration {
public:
};
```

2. The **`loadConfiguration()`** and **`saveConfiguration()`** methods of **DmocMotorController** and **MotorController** can be used as a reference for how to load and save configurations.

3. PrefHandler handles the memcache for various devices. Each device needs its own unique name and id number. These are held in the file **`DeviceTypes.h`**

```
enum DeviceId { //unique device ID for every piece of hardware possible
      DMOC645 = 0x1000,
      BRUSA_DMC5 = 0x1001,
      CODAUQM = 0x1002,
      BRUSACHARGE = 0x1010,
      TCCHCHARGE = 0x1011,
      LEAR=0x1012,
      THROTTLE = 0x1030,
      POTACCELPEDAL = 0x1031,
      POTBRAKEPEDAL = 0x1032,
      CANACCELPEDAL = 0x1033,
      CANBRAKEPEDAL = 0x1034,
      EVICTUS = 0x4400,
      ICHIP2128 = 0x1040,
      DCDC = 0x1050,
      THINKBMS = 0x2000,
      SYSTEM = 0x5000,
      HEARTBEAT = 0x5001,
      MEMCACHE = 0x5002,
      PIDLISTENER = 0x6000,
      ELM327EMU = 0x650,
      INVALID = 0xFFFF
};
```

Here, for the EVIC class we call the device `EVICTUS` and assign the number `0x4400` to it.

In our object file **`EVIC.cpp`**, we have to alter our device constructor to include this.

```
EVIC::EVIC() : Device()
{
    prefsHandler = new PrefHandler(EVICTUS);
    commonName = "Andromeda Interfaces EVIC Display";
}
```

This registers us for the PrefsHandler to provide memcache space. We can then use LoadConfiguration and SaveConfiguration to handle our persistent data that we want to use from session to session without loss.

MULTIPLE DEVICES ON CAN

GEVCU provides two independent CAN ports that can be configured for different speeds and operated more or less simultaneously. The nature of CAN is that a variety of hardware devices can be on the bus simultaneously as long as they all use the same data speed. And so of course a variety of object modules is envisioned, each servicing one or more CAN devices.

Again to accommodate this without chaos, we have developed a CAN handler. The CAN handler monitors the two CAN busses and when a CAN message is received, the message is passed to the appropriate module. And so each module performing CAN tasks must be registered with the CAN handler as to which specific CAN message IDs it wants to receive.

A canbus handler allows your device to register to receive frames that match the mask. The call to attach has four parameters. The first will basically always be "this". The second is the ID to match, the third is the mask, and the fourth specifies whether you are interested in standard (false) or extended (true) frames.

```
canHandlerEv->attach(this, 0x230, 0x7f0, false);

canHandlerCar->attach(this, 0x650, 0x7f0, false);
```

In this case, we are registering with CAN bus 0 (Ev) to receive any message with an id between **0x230** and **0x23F**.

We are also registering with CAN bus 1 (Car) to receive any CAN message with ID **0x650** through **0x65F**.

If you need a quick refresher on masks and IDs: CAN bus filtering happens by taking every incoming frame and doing a bitwise AND with each registered mask in turn. The result of this is then compared to the associated ID for that mask. If the two match then your device will receive that frame. For instance, if a frame with **0x23D** were to come in and the **DmocMotorController** had registered a mask of

0x7F0 and an ID of **0x230. 0x23D AND 0x7F0 = 0x230**. This does match the ID and so **DmocMotorController** would get the frame.

Your class should have a **handleCanFrame()** method if you have registered one or more canbus handlers. This method takes a pointer to a CAN_FRAME structure as input.

```
void DmocMotorController::handleCanFrame(CAN_FRAME *frame)
{

}
```

To send a frame, first create a CAN_FRAME structure variable, fill it out, and use **sendFrame()** to send it.

```
CAN_FRAME output;  //establishes variable output of structure CAN_FRAME
output.length = 8;  //Sets number of bytes in payload
output.id = 0x207;   //Sets message id
output.extended = 0; //0 standard 11 bit frame 1 for extended 29bit frame
output.data.bytes[0] = 0xa5; //This is simply puts value 5A
output.data.bytes[1] = 0xa5; //in the first three payload bytes
 output.data.bytes[2] = 0x5a;
 output.data.bytes[3] = 0x00;
 output.data.bytes[4] = 0x00;
 output.data.bytes[5] = 0x00;
 output.data.bytes[6] = 0x00;
 output.data.bytes[7] = 0x00;
CanHandler::getInstanceEV()->sendFrame(output);  //Send the frame
```

IN this case we establish the variable output as type CAN_FRAME

We set the length to 8 bytes and the message ID to 0x207. We set it for standard 11bit frames and then populate the eight bytes with whatever we like. Then call the can handler to send frame OUTPUT. On the EV (CAN0) bus.

So note that we used a call to the can handler to register our ids and masks for incoming messages, and provided a method to handle those incoming messages by the specific name **handleCanFrame**.

We used another call to the canhandler to send a message we had preloaded into a CAN_FRAME variable.

In this way, all traffic is handled by the can handler and it can intelligently parse out CAN frames to hungry object modules waiting for them , and also stream messages onto the bus without collision or panic party.

The GEVCU project is designed such that the core system can be extended with modules that provide support for various pieces of hardware. Designing a new module is not conceptually complex but does require several things to be successful.

Note about formatting: **BOLD** words are class names. *Italic* words are file names. Underlined words are methods or members of a class. Methods are postfixed with ().

1. The first order of business is to find an existing module of hopefully the same basic type as your desired new module. For instance, if you hope to add support for an XYZ inverter then maybe you'd like to copy the **DmocMotorController** class. Rename the files with the same format. Maybe you'd name the files *XYZMotorController.cpp* and *XYZMotorController.h* If you cannot find an existing device of the same type then still use an existing set of files. The **DmocMotorController** class could be used as the basis for most any other device.
2. Rename all of the class references in the copied files as well.
3. Notice that **DmocMotorController** inherits from **MotorController** and **CanObserver**. If you are not creating a motor controller class then inherit from something else. Currently the other good choices are **Throttle** or **Device**. **Device** is always a decent choice if no more specific parent class exists. Inheriting from **CanObserver** is required if your device needs access to either of the CAN busses.
4. Each device can store data within EEPROM. Each device should create a configuration class that inherits from **DeviceConfiguration**. The loadConfiguration() and saveConfiguration() methods of **DmocMotorController** and **MotorController** can be used as a reference for how to load and save configurations.

5. Each device also must have a unique ID and should set what type of device it is. These are set with getId() and getType(). There is a global list of IDs that you should add your device to. *DeviceTypes.h* contains both the allowable device types and the ID of each and every device that is compiled into the firmware. You will note that **DmocMotorController** does not contain getType(). Instead it maintains the parent class version since it is still a motor controller.
6. Next, edit the constructor. Most of the constructor will be custom to your device but you will need to set prefsHandler to use your device ID constant instead of DMOC645. Also, set commonName to a nice, short description of your device.
7. Then there are several edits to make in the setup method. As shown in the **DmocMotorController** class, one should set up the tick handler, load the EEPROM configuration, call the parent setup (if applicable), and set up canbus handlers (if applicable).
    a. A tick handler allows your device to receive periodic ticks so that it can do processing at set intervals. This might be so that you can send canbus frames or so that you can check for input on one or more pins.
    b. Your class should have a handleTick() method if you have registered a tick handler.
    c. A canbus handler, handleCanFrame(), allows your device to register to receive frames that match the mask. The call to attach has four parameters. The first will basically always be "this". The second is the ID to match, the third is the mask, and the fourth specifies whether you are interested in standard (false) or extended (true) frames. If you need a quick refresher on masks and IDs: Canbus filtering happens by taking every incoming frame and doing a bitwise AND with each registered mask in turn. The result of this is then compared to the associated ID for that mask. If the two match then your device will receive that frame. For instance, if a frame with 0x23D were to come in and the **DmocMotorController** had registered a mask of 0x7F0 and an ID of 0x230. 0x23D AND 0x7F0 = 0x230. This does match the ID and so **DmocMotorController** would get the frame.

    d. Your class should have a handleCanFrame() method if you have registered one or more canbus handlers. This method takes a pointer to a canbus frame as input.
8. As covered above, handleTick() and handleCanFrame() will form the majority of the action in most devices. Many devices will want to be able to send canbus frames out as well. This is done through CanHandler::getInstanceEV() or CanHandler::getInstanceCar(). The **MotorController** class pre-caches CanHandler::getInstanceEV() into the canHandlerEv member. To send a frame, first create a CAN_FRAME variable, fill it out, and use sendFrame() to send it. The DMOC sendCmd1 method shows how to do this.
9. If your device handler needs access to physical IO then use the systemIO member variable inherent to all devices. This variable references the **SystemIO** class which has the ability to interact with all of the digital I/O and analog inputs.
10. Now that the device handler has been constructed it is time to connect it with the rest of the system. In *GEVCU.ino* there is a function called createObjects(). You will need to instantiate your class here. You will also have you add your .h file as an include in *GEVCU.h*. With this done everything should automatically work. The system uses the **PrefsHandler** instance you setup in the class constructor to determine whether a given device is enabled and only sets up devices which are enabled.
11. A consequence of above is that you *must* enable your new device before it will do anything. This is done by typing "ENABLE=" followed by the ID of your new device.

1. Create your object.cpp and object.h files for your class. Often it is of help to copy an existing file set and rename them. If your new object is a motorcontroller, for example, use dmocMotorController.cpp and dmocMotorController.h. If it is a charger, copy a charger module.
2. List your device in DeviceTypes.h
3. Add your device as an include in GEVCU.h

4. Revise the createObjects section of GEVCU.ino to instantiate your class there.