**GEVCU Project**

How to Develop a New Module

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).
   1. 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.
   2. Your class should have a handleTick() method if you have registered a tick handler.
   3. 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.
   4. 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.