OVMS

Open Vehicle Monitoring System

www.openvehicles.com

OVMS Development
Guide v2.5.2 (1st January 2017)
For OVMS Hardware Module v2 and Firmware v2.x.x

History

v2.5.2	1st January 2017	Added new car model variables
v2.5.1	16th August 2013	Make car_tbattery signed integer
v2.3.0	30th April 2013	Extensions
v2.0.1	28th January 2013	Extensions and clarifications
v2.0.0	9th January 2013	Initial version written

WELCOME 5

DEVELOPMENT OVERVIEW 5

VEHICLE FIRMWARE DEVELOPMENT TOOLS 6

DEVELOPMENT ENVIRONMENT 6
DEVELOPMENT TOOLS 6
GITHUB 10
COMPILE AND FLASH YOUR FIRST FIRMWARE 10
CHIPS USED 11
ARCHITECTURE AND TOOL HINTS 11

VEHICLE MODULE DIAG PORT 13

THE DIAG PORT 13
USING THE DIAG PORT FOR DEBUG LOGGING 14

VEHICLE FIRMWARE OVERVIEW 15

CODE MODULES 15
THE POLLING MAIN LOOP 16
LED CONTROL 16

OVMS VEHICLE MODULES 17

OVMS V2 AND PLUG-IN VEHICLE MODULES 17
VEHICLE.H AND VEHICLE.C 17
IMPLEMENTING A NEW VEHICLE MODULE 19
A NOTE ON VEHICLE ROM CODE 21
A NOTE ON VEHICLE RAM DATA 22
INITIALISING THE CAN BUS 23
TRANSMITTING ON THE CAN BUS 24

THE OVMS VIRTUAL VEHICLE 26

WHAT IS THE OVMS VIRTUAL VEHICLE? 26
GLOBAL DATA STORAGE 26

Vehicle Identification....26
GPS STATUS 27
Tire Pressure Monitoring System....28
Driving Status....29
Vehicle Environment....29
Battery Status....32
Charging Status....32
VEHICLE MODULE DEVELOPMENT CHECKLISTS 36
DEVELOPMENT CHECKLISTS 36
VEHICLE-SPECIFIC EXTENSIONS 38
NET_MSG COMMAND HANDLERS 38
NET_SMS SMS HANDLERS 38
NET_SMS SMS EXTENSIONS 39
OVMS SERVER DEVELOPMENT 40
OVMS APP DEVELOPMENT 41
CONCLUSIONS 42

Welcome

The OVMS (Open Vehicle Monitoring System) team is a group of enthusiasts who are developing a means to remotely communicate with our cars, and are having fun while doing it.

The OVMS module is a low-cost hardware device that you install in your car simply by installing a SIM card, connecting the module to your car's Diagnostic port connector, and positioning a cellular antenna. Once connected, the OVMS module enables remote control and monitoring of your car.

This document presents an overview of the development tools and techniques you will need to work on the OVMS system. You might be extending what OVMS does already (either for general consumption, or for your own private use), adding support for a new vehicle, or using the OVMS framework to implement something completely different. Whatever your purpose, please remember that OVMS is an Open Source project without restrictions, that has got to where it is today by the contributions of so many; so please try to share what you yourself do with it.

Development Overview

OVMS development can be divided into three main parts:

1. Vehicle Firmware Development – working on the module inside vehicles.
2. Server Development – working with the OVMS server code.
3. App Development – working with the remote Apps that access the vehicle information.

While some information presented here is specific to a particular area of development, in general we try to cover all three parts.

Vehicle Firmware Development Tools

Development Environment

The development environment of choice is the Microchip MPLAB X IDE and MPLAB C18 compiler. This is available for free on Linux, Mac and Windows – but be aware that the Windows C18 compiler is restricted in that optimisations do not work correctly unless you purchase the commercial version of the compiler. For that reason, we recommend either Linux or Mac for development.

http://www.microchip.com/pagehandler/en-us/family/mplabx/

You will need a JAVA runtime, MPLAB X IDE and C18 (not XC8) compiler. Note that Microchip is migrating to the XC range of compilers, but OVMS uses the Microchip C18 compiler – you may need to hunt a bit on the Microchip site to find it, but it will be there.

In addition, you may find the free Microchip CAN timing calculator useful:

http://intrepidcs.com/support/mbtime.htm

Development tools

You will need the following tools:

- An OVMS v2 module

OVMS v2 Module (on fasttech.com)

- A compatible GSM antenna

OVMS GSM Antenna (on fasttech.com)

- A compatible GPS antenna (optional, depends on vehicle type)

OVMS GPS Antenna (on fasttech.com)

- A compatible 2G/3G SIM (with data)

  OVMS uses 2G GPRS data, but will work with 3G SIMs. You will need to obtain this from a local cellular provider. There are too many to list here, but this forum thread on TMC should give you some pointers:

  OVMS Installation thread (on TMC)

  Alternatively, you can use the hassle-free GeoSIM that OVMS has partnered with:

Hassle-free SIMs with GPRS data for OVMS

- An OVMS vehicle-specific cable

OVMS Tesla Roadster Vehicle Cable (from fasttech.com)

OVMS Generic OBDII Vehicle Cable (from fasttech.com)

You can either make this up yourself (the wiring is pretty simple – just 12V power, GND, CAN-L and CAN-H) or use either the Tesla Roadster or Generic OBDII cables.

- A Microchip firmware flash tool (PICKIT3 recommended for development)

Microchip PICKIT2 clone (from fasttech.com)

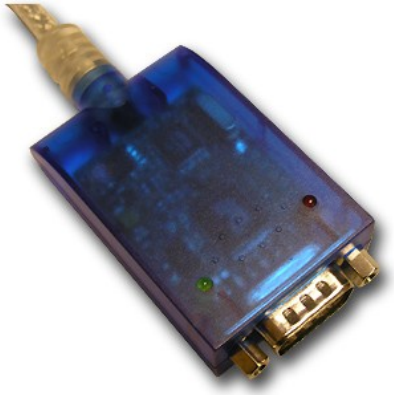
Microchip PICKIT3 clone (from fasttech.com)

- A USB/RS232 adaptor cable (or RS232 port on your computer)

  You will need a way of connecting your computer to the DB9 RS232 DIAG port on the OVMS module. If your laptop already has a RS232 port, then you can connect directly, otherwise you will need to source a USB/RS232 adaptor.

  Search for USB RS232 (on fasttech.com)

- A USB/CAN bus adaptor, or USB/OBDII dongle (optional, but required for CAN bus message reverse engineering)



  Example USB/CAN adaptor (from canusb.com)

Example OBDLINK-SX (from scantool.net)

GITHUB

The source code for OVMS vehicle firmware is publicly available on a system called GITHUB. This is a cloud server hosting projects using the GIT source code revision control system.

https://github.com/markwj/Open-Vehicle-Monitoring-System

You should create yourself an account on github, then use the FORK function to get yourself a copy of the project into your own repository. You will want to download your own forked repository from github to your local computer.

Detailed operation of GIT and GITHUB is beyond the scope of this document, but a general summary of the recommended method of working is that you will fork yourself your own clone of the master OVMS GITHUB project and then work on your own fork. From time to time, you can refresh your copy with the upstream master. You can also submit your improvements / extensions back to the upstream master by a PUSH request (or merely creating a patch and submitting it for review and inclusion with the master project).

Compile and Flash your First Firmware

A simple summary is:

1. Start MPLAB X IDE.

2. Open the OVMS.X project from your OVMS github fork (under 'vehicle' directory).

3. Select configuration "V2 Experimental". Right click on the project, to get to project settings, and customize it for the flash tool you are using (eg; pickit2 or pickit3). You will probably want to choose the "Preserve device EEPROM" setting, to avoid overwriting your vehicle settings in EEPROM every time you flash the device.

4. Click "Clean and Build" and ensure the compile completes successfully.

5. Connect your PICKIT to the vehicle module, choosing 5V power from the programmer (or power it directly from 12V via the vehicle port). Then, click "Make and Program Device" to flash the device with your code.

Congratulations, you just did your first firmware build & flash!

Chips Used

The OVMS v2 module uses the following chips:

- Microcontroller: Microchip PIC18F2685

  http://www.microchip.com/wwwproducts/Devices.aspx?dDocName=en026328

- Cellular Modem & GPS: SIMCOM SIM908

  http://www.simcom.us/product_detail.php?cid=1&pid=38

Architecture and Tool Hints

- MPLAB X needs a restart from time to time... e.g. if suddenly the editor highlights strange missing symbols, or if the editor refuses to save some files...

- PIC18F implements a harvard architecture = RAM and ROM use different addressing schemes.
  - String constants are ROM, so need to be copied over into some RAM buffer before using them as RAM * arguments. Use e.g. strcpypgm2ram() for this. (pgm = program memory)
  - sprintf etc. use ROM templates and RAM arguments.

    - Always look for special lib functions using name extensions like "pgm2ram", these should be used if possible (optimized).

- PIC18F is little endian, i.e.
    - **int 0x1234 = { 0x34, 0x12 }**
    - **long 0x11223344 = { 0x44, 0x33, 0x22, 0x11 }**
    - Bit fields are packed right to left

- The stack is small, really. The call stack allows for just 31 levels, and the software stack is limited to one bank = 256 Byte.
    - Be aware that ISR code must be callable in any application depth, and the vehicle CAN ISR will need at least two call levels (due to the framework).
    - Avoid deep nesting of calls, try to inline code or use macros as often as possible!
    - Avoid complex and large arguments to calls! This especially applies to varargs calls like sprintf() – these will not produce a stack overflow, but you will get really strange behaviour...
    - The debugger is no real help in stack problems. See below for stack overflow debug code.
    - Bottom line is: KISS – keep it small & simple

- PIC18F has 3328 Bytes of RAM. You know what that means, don't you ;-)
    - To be precise, it's SRAM. So it will keep content over a reset, even over a short power down. To make use of this, you need to define your static/global vars undefined. You can use the car_type[] variable to distinguish between init and reset situations (see vehicle initialisation).

- PIC18F has no FPU (floating point unit). All float operations are done in software, so you better avoid using them in time critical situations... like interrupt handlers for example.

- C18 is not ISO standard compliant in some important ways! See user guide for details.

- C18 is configured to "no integer promotions" for the OVMS firmware. That means "byte * byte = byte" / "int * int = int" etc., so you need explicit type casts at some unexpected places!

- The C18 library printf/scanf family does not support float/double arguments! atof() also is not functional. If you need float parsing / output you have to implement it. For example by scaling integers:

- // **Format scaled int into two ints for sprintf**
  // **...example: sprintf(buf, "%d.%02u", FMTINT(val,100))**
  #define FMTINT(f,d) (f) / (d), ((f) >= 0) ? ((f) % (d)) : (-(f) % (d))
  #define FMTUINT(f,d) (f) / (d), (f) % (d)

- C18 supports bit fields of only max 8 bits length

- C18 is a dumb compiler. If you write too complex expressions, especially with floats involved, it will fail without telling you, silently producing broken code.
  - Example: This is too complex for C18:
    stddev = sqrt(((float)sqrsum/n) - SQR((float)sum/n));
    ...as a result, sprintf() will fail somewhere else in the code...
    ...resolving was:
    float f = ((float)sqrsum/n) - SQR((float)sum/n);
    stddev = sqrt(f);

- C18 is also not very good at optimizations. It will especially completely fail to do any kind of loop unrolling, even factoring out simple constant expressions will fail in many (most?) cases.
- It will also fail to eliminate unused local variables and unused functions.

- Bottom line: Code simple and optimize yourself.

Stack depth debugging

The software stack (function parameters and local variables) is maintained in register FSR1, the hardware call stack usage can be examined in register STKPTR.

The deepest call depth normally will occur in your CAN ISR code. Introduce some global variables like...

```
UINT MAX_FSR1 = 0;
UINT8 MAX_STKPTR = 0;
```

...then add this to your ISR(s):

```
if (FSR1 > MAX_FSR1)
  MAX_FSR1 = FSR1;
if ((STKPTR&0x1f) > MAX_STKPTR)
  MAX_STKPTR = (STKPTR&0x1f);
```

Now you can inspect or output the MAX_* vars to check your stack usage.

Vehicle Module DIAG Port

The DIAG port

The OVMS v2 module includes a port labeled DIAG. This port offers an RS232 compatible tap into the communications link between the PIC microcontroller and SIMCOM modem. It can be used to monitor these communications, as well as issue arbitrary commands. As it is a tap, either the commands from the PIC microcontroller, and/or SIMCOM modem, can be spoofed for diagnostics, debugging and testing. The OVMS firmware also contains a special DIAG SETUP mode that can be accessed via this port. The port is not used for anything during normal day-to-day operation, and is only there for development and diagnostics purposes.

To connect to the DIAG port:

- Connect your PC to the DIAG port using a standard serial cable or serial to USB adaptor. Communication parameters are 9600 8N1, no handshake.
- You will need a terminal emulator on your PC. For Linux/Mac, you can use cu or screen. For Windows, you can use Minicom, or any of its more modern replacements.
- Note: The OVMS module needs +12V power input to function properly. It may appear to power up at +5V from the PICKIT flash tool, but the modem side really requires +12V to operate correctly. This also applies to the DIAG port. The best approach is to power the module through the vehicle port.

As the DIAG port taps the communication between the microcontroller (PIC18F2685) and the GPRS/GPS modem (SIM908), you can see the modem output and send commands to it any time. You can also see the microcontroller output, but only indirectly via the modem's command echo. During normal boot

up modem initialization, OVMS disables the modem's echo mode, so you need to enable it again by sending "ATE<CR>" (you might consider assigning that to a macro key in your terminal emulator).

The OVMS firmware also supports a special DIAG boot mode (if compiled in) that allows you to configure the module and test commands via the DIAG port. To boot into DIAG mode, just send the magic word "SETUP<CR>" to the module while it's booting up (you might consider assigning that to the next macro key). The timing for how/when this is done is fairly critical, so we have prepared a little youtube video on this:

http://www.youtube.com/watch?v=-24PoWtHVzk

So you've got about 20 seconds after powering the module / reset to send the magic word. You may send the word as often as you want, the module will normally respond to it after the first modem init (i.e. after "Call Ready" / "GPS Ready"). If it does not, reboot.

Once you're in DIAG mode, enter:

- "HELP" to list available commands (also lists SMS and MSG commands)
- "DIAG" to see a short status overview
- "S command" to execute an SMS command
- "M command" to execute a MSG command

In DIAG mode, all SMS and MSG outputs are sent to the DIAG port instead of the GPRS network. MSG output will not be encrypted.

Using the DIAG Port for debug logging

Besides implementing some sort of "DEBUG" command, you can also use the DIAG port for direct debug outputs. To do so, just send a comment to the modem: "# hello world!\r\n". After enabling echos you will see the output on your terminal.

Be aware that your debug outputs will be sent back to the microcontroller comm port and might confuse some ongoing request / response protocol. So, this kind of debug log should only be used during development, not in a production build. Good style is to check for an empty receive buffer before, so your outputs will not disturb a modem response and/or diag command input. Code example:

If ((net_buf_mode == NET_BUF_CRLF) && (net_buf_pos == 0))

```
{
sprintf( net_scratchpad, "# tic1: ss=%d\r\n", can_batt_sensors_state );
net_puts_ram( net_scratchpad );
}
```

Vehicle Firmware Overview

Code Modules

The vehicle firmware code is split into modules, with each module in its own source code file (usually, by convention, <module>.h and <module>.c). The modules are:

- Primary Modules:
    - ovms
      Main module implementing the initialization process and main loop.
    - net
      Network/Modem communications layer (state machine)
    - net_sms
      Standard framework SMS commands and utils
    - net_msg
      Standard framework MSG commands and utils (network protocol)
    - vehicle
      Vehicle plug-in framework controller
- Auxiliary modules:
    - params
      EEPROM read/write utils (storing modem config parameters)
    - inputs
      Hardware-independent input framework (internally used)

    - led
      Hardware-independent LED framework (internally used)

    - utils
      General utility library

    - diag
      DIAG/SETUP mode framework

- Vehicle Modules

    - vehicle_none
      Default (stub) vehicle plug-in module

    - vehicle_*
      Other vehicle plug-in modules are named with the vehicle type (e.g.; vehicle_teslaroadster, vehicle_twizy, etc)

- Support modules:

    - UARTIntC
      Support library implementing UART communications

    - crypt_base64
      Support library implementing base64 encoding and decoding

    - crypt_hmac
      Support library implementing HMAC digests

    - crypt_md5
      Support library implementing MD5 digests

    - crypt_rc4
      Support library implementing RC4 cryptography

The Polling Main Loop

After boot time initialization, the main loop controls initialization and polling for each module in the firmware. Typically, each module implements a _initialise() function (e.g. net_initialise(), vehicle_initialise(), etc) called during initialization, and a _ticker() function (e.g. net_ticker(), vehicle_ticker(), etc) called once a

second. In addition, there are _ticker10th() hooks for 1/10th second tickers, and a _poll() hook to tell the net module that there is modem data ready and waiting.

LED Control

The LEDs are controlled by the led module, with simple calls to set specific status (green) and fault (red) codes. The module is implemented in that the caller just requests a status or fault code, and the module itself will control the flashing of the LEDs from that point on (until the status or fault code is changed).

OVMS Vehicle Modules

OVMS v2 and Plug-In Vehicle Modules

Version v2 of the OVMS firmware introduces support for a plug-in architecture for vehicle modules. With this architecture, support for multiple vehicles can be included in a single vehicle firmware, switched at run-time by module parameter.

vehicle.h and vehicle.c

The code files vehicle.h and vehicle.c implement the plug-in vehicle module system. Let's look at the shared variables delivered by this framework:

- unsigned int can_granular_tick

  This is an internal ticker used to generate 1min, 5min, etc, ticker calls – it can be read and used in the _ticker1() hook to implement other forms of granular ticker.

- unsigned char can_datalength
  unsigned char can_databuffer[8]

  These variables are typically used to store the number of valid can bytes, and the bytes themselves, by vehicle module _poll0() or _poll1() reception of CAN data hooks. They are for the use of the vehicle module.

- unsigned char can_minSOCnotified

  The minSOC notified flag – used by vehicle module to implement notification when SOC falls below a defined level.

- unsigned char can_mileskm

  A flag to indicate whether the user has requested miles or kilometers as their preferred units (to save the vehicle module having to look this up in a parameter – especially during handling of a CAN interrupt). It is set by the framework to either 'M' or 'K'.

- unsigned char* vehicle_version

A pointer to be set by the vehicle module to a version string to indicate the version of the vehicle module code.

- unsigned char* can_capabilities

  A pointer to be set by the vehicle module to a capability string to indicate the capabilities of the vehicle mode code.

In addition to the global variables, there are a number of hooks. These can be set, during vehicle module initialization, to allow the vehicle module to hook into specific system events. The convention is that if a function is not NULL then it is called at the appropriate time. If it returns TRUE then it is considered to have completed the operation and the default action should not be run at all. If it returns FALSE then it is an indication that the default action should run as well.

- BOOL (*vehicle_fn_init)(void)

  The module initialization function.

- BOOL (*vehicle_fn_poll0)(void)

  The CAN RX callback function to receive CAN bus data on channel 0.

- BOOL (*vehicle_fn_poll1)(void)

  The CAN RX callback function to receive CAN bus data on channel 1.

- BOOL (*vehicle_fn_ticker1)(void)

  A callback hook called once per second.

- BOOL (*vehicle_fn_ticker10)(void)

  A callback hook called once every 10 seconds.

- BOOL (*vehicle_fn_ticker60)(void)

  A callback hook called once every 60 seconds.

- BOOL (*vehicle_fn_ticker300)(void)

  A callback hook called once every 300 seconds.

- BOOL (*vehicle_fn_ticker600)(void)

  A callback hook called once every 600 seconds.

- BOOL (*vehicle_fn_ticker)(void)

  A callback hook called once every second, as part of the main framework.

- BOOL (*vehicle_fn_ticker10th)(void)

  A callback hook called 10 times every second.

- BOOL (*vehicle_fn_idlepoll)(void)

  A callback hook called very often (typically whenever the main loop is idle and not calling other callbacks and is idle waiting for a timer or incoming data).

- BOOL (*vehicle_fn_commandhandler)(BOOL msgmode, int code, char* msg)

  A callback hook for incoming net_msg protocol commands. This can be used to override or implement standard and vehicle specific commands.

- BOOL (*vehicle_fn_smshandler)(BOOL premsg, char *caller, char *command, char *arguments)

  A callback hook for incoming net_sms SMS commands. This can be used to override or implement standard and vehicle specific SMS commands.

- BOOL (*vehicle_fn_smsextensions)(char *caller, char *command, char *arguments)

  A callback hook for incoming net_sms SMS commands. This can be used to supplement standard and vehicle specific SMS command responses with additional information.

Implementing a New Vehicle Module

It is generally best to implement a new vehicle module in co-operation with the core OVMS maintainers. They can assist you with a consistent naming

convention, and changing the master github project to include a stub of your vehicle code.

To implement a new vehicle module, you will need the following:

1. A short descriptive name of the vehicle, as the vehicle_* file name (e.g.; twizy, teslaroadster, thinkcity, etc).
2. A short descriptive upper-case #define for the vehicle module. This is typically the same (or similar) to #1.
3. A 2 character type code for the vehicle (e.g.; RT, TR, TC, etc).
4. A 2 character subtype code for specific vehicle modules (e.g. 1S for Tesla Roadster v1.x Sport, 2N for Tesla Roadster v2.x Non-Sport, etc).

Let's imagine a car called the Futura-Electric, with three versions (10kWh, 20kWh and 40kWh batteries). We are going to use one vehicle module for all three versions (as they are so similar), and find the actual version by looking at messages on the CAN bus. For this car, we're going to use short descriptive name "futura", #define "#FUTURA", 2 character vehicle type code "FE", and 2 character subtype code 10, 20 or 40.

Now, to implement this vehicle in OVMS:

1. We edit ovms.dev and include:

   //**#define OVMS_CAR_FUTURA**

   This lets us know we have a new vehicle type, but it is commented-out so it is not included in all firmwares. We're going to let the vehicle firmware configuration itself include it.

2. Edit the project properties, configurations, C compiler pre-processor directives, to include OVMS_CAR_FUTURA in the "V2 experimental" and "V2 production" builds.

3. Create a blank vehicle_futura.c file in the project.

4. In the project explorer, choose each configuration one-by-one, right-click on vehicle_futura.c and exclude it from the build in all configurations except the ones we've #defined it in ("V2 experimental" and "V2 production").

5. Change vehicle.c – just above the "#ifdef OVMS_CAR_NONE", include a new block for our new vehicle:

```
#ifdef OVMS_CAR_FUTURA
  else if (memcmppgm2ram(p, (char const rom far*)"FE", 2) == 0)
    {
    void vehicle_futura_initialise(void);
    vehicle_futura_initialise();
    }
#endif
```

6. Implement the stub of the vehicle in vehicle_futura.c as:

```
#include <stdlib.h>
#include <delays.h>
#include <string.h>
#include "ovms.h"
#include "params.h"

#pragma udata overlay vehicle_overlay_data

#pragma udata

BOOL vehicle_futura_initialise(void)
  {
  char *p;

  car_type[0] = 'F'; // Car is type NONE
  car_type[1] = 'E';
  car_type[2] = 0;
  car_type[3] = 0;
  car_type[4] = 0;

  net_fnbits |= NET_FN_INTERNALGPS;   // Require internal GPS

  return TRUE;
  }
```

Then, Make Clean and Build… We should end up with a new vehicle module that does nothing but uses the OVMS internal GPS to at least report the vehicle location. You can even now configure the module with the SMS MODULE command, to choose vehicle type “FE”, and then SMS “GPS?” to show the current GPS position.

As you can see, there are quite a few steps to be done, and you must co-operate with other vehicles in the system. This is why it is generally best to implement a new vehicle module in co-operation with the core OVMS maintainers. They can assist you with a consistent naming convention, and changing the master github project to include a stub of your vehicle code.

A note on Vehicle ROM code

All plug-in vehicle modules included in the same firmware must share the limited ROM (code) space of the PIC18F2685 microcontroller. This means you must take care with how you code – keep things clean and simple. At the moment, we can include all vehicle modules in the same firmware, but this may change longer-term and we may include different sets of vehicles in different firmware (e.g. firmware #1 for RT, TR, TC, firmware #2 for O2, VA, FE, etc).

We can support an unlimited number of plug-in vehicle modules in the firmware, but may need to have multiple firmware sets to support dozens of different vehicles.

A note on Vehicle RAM data

All plug-in vehicle modules included in the same firmware must share the limited RAM space of the PIC18F2685 microcontroller. As we only have 3,328 bytes of RAM, this could be a major problem. If vehicle #1 uses 50 bytes, vehicle #2 200 bytes, vehicle #3, 150 bytes – the total would be 800 bytes! That would severely limit the number of vehicles that can be supported.

However, there is a neat trick to avoid this. Only one vehicle module can be running at any one time, so we can put all the vehicle module data in the same place. That is the purpose of the

```
#pragma udata overlay vehicle_overlay_data
```

directive that you saw in the sample code above. Any variables declared after that directive, but before the "**#pragma udata**" directive that follow, will be put in the vehicle_overlay_data section of ram – *and each vehicle's data will be put at the same place* in ram! It behaves like a C UNION, split over different source files. Each vehicle gets its own space to store its data, but the total amount of ram allocated is only equal to the largest amount of ram that the biggest module requires. In our example above, vehicle #2 requires 200 bytes of ram and that will be the size of vehicle_overlay_data. The result behaves like this:

```
union
 {
 … vehicle1_data …;
 … vehicle2_data …;
 … vehicle3_data …;
 }
```

The drawback is that this overlayed data structure cannot be initialized by the C runtime (how could it initialize it if vehicle #1 required a different initialization to vehicle #2?). In practice, you should initialize overlayed data yourself in the _initialise() function that is called during module initialization (which is when you are the selected vehicle).

As the PIC18F microprocessors limit each udata section to a maximum of 256 bytes, there is a second overlay section also available, to allow each vehicle to use up to 2x256 = 512 bytes of overlay data:

```
#pragma udata overlay vehicle_overlay_data2
```

Initialising the CAN bus

You should initialize the CAN bus however you require during module initialization. A typical implementation of this is:

```
CANCON = 0b10010000; // Initialize CAN
while (!CANSTATbits.OPMODE2); // Wait for Configuration mode

// We are now in Configuration Mode
RXB0CON = 0b00000000;       // RX #0 mask and filters
RXM0SIDL = 0b10100000;
RXM0SIDH = 0b11111111;
RXF0SIDL = 0b00000000;
RXF0SIDH = 0b00100000;

RXB1CON = 0b00000000;       // RX #1 mask and filters
RXM1SIDL = 0b11100000;
RXM1SIDH = 0b11111111;

BRGCON1 = 0x01; // SET BAUDRATE to 500 Kbps
BRGCON2 = 0xD2;
BRGCON3 = 0x02;
```

```
CIOCON = 0b00100000; // CANTX pin will drive VDD when recessive
if (sys_features[FEATURE_CANWRITE]>0)
 {
 CANCON = 0b00000000;  // Normal mode
 }
else
 {
 CANCON = 0b01100000; // Listen only mode, Receive bufer 0
 }

// Hook in...
vehicle_fn_poll0 = &vehicle_futura_poll0;
```

Note that we are using RXB0 (RX channel #0) and hence the vehicle_fn_poll0() hook for the framework to call us (during the can rx interrupt handler, so take care).

Also note the use of sys_features[FEATURE_CANWRITE]>0 to control whether the CAN bus is to be initialized in normal or listen-only mode.

The vehicle_futura_poll0() function looks like this:

```
BOOL vehicle_futura_poll0(void)
 {
 can_datalength = RXB0DLC & 0x0F; // number of received bytes
 can_databuffer[0] = RXB0D0;
 can_databuffer[1] = RXB0D1;
 can_databuffer[2] = RXB0D2;
 can_databuffer[3] = RXB0D3;
 can_databuffer[4] = RXB0D4;
 can_databuffer[5] = RXB0D5;
 can_databuffer[6] = RXB0D6;
 can_databuffer[7] = RXB0D7;

 RXB0CONbits.RXFUL = 0; // All bytes read, Clear flag

 // Implement the handling of the incoming data in can_databuffer
 // (length can_datalength) here…

 return TRUE;
 }
```

You can refer to the PIC18F2685 data sheet and other vehlcle_* module source code for examples of how to manage the data coming in the CAN bus.

Transmitting on the CAN bus

We transmit on the CAN bus, typically in _ticker*() functions. We do not generally recommend doing this in the _poll*() functions, as that is in the CAN reception interrupt handler.

The code to transmit looks like this:

```
if (sys_features[FEATURE_CANWRITE]>0)
  {
  while (TXB0CONbits.TXREQ) {} // Loop until TX is done
  TXB0CON = 0;
  TXB0SIDL = (arb_id & 0x07)<<5;
  TXB0SIDH = (arb_id>>3);
  TXB0D0 = 0x01;
  TXB0D1 = 0x02;
  TXB0D2 = 0x03;
  TXB0D3 = 0x04;
  TXB0D4 = 0x05;
  TXB0D5 = 0x06;
  TXB0D6 = 0x07;
  TXB0D7 = 0x08;
  TXB0DLC = 0b00001000; // data length (8)
  TXB0CON = 0b00001000; // mark for transmission
  }
```

Note that before transmitting, you should use the loop on **TXB0CONbits.TXREQ** to ensure the controller is ready to transmit.

Also note that you should only transmit if sys_features[FEATURE_CANWRITE] is >0. The reason for this is that the CAN bus will be in listen-only mode if this is ==0. In that case, you can't transmit, so **TXB0CONbits.TXREQ** will never go true, and you will loop forever (or until the hardware watchdog timer fires and the module is rebooted).

The OVMS Virtual Vehicle

What is the OVMS Virtual Vehicle?

OVMS supports multiple vehicles by implementing a virtual vehicle. The parameters and behavior of each real vehicle are mapped to this virtual vehicle, and that is what is used by the Apps.

The plug-in vehicle modules typically read data from the vehicle's CAN bus, and update the global data storage values of the virtual vehicle, to reflect that state. They convert the physical vehicle status to virtual vehicle status.

In addition, the plug-in vehicle modules should issue notifications whenever certain events occur (such as a charge being interrupted).

Global Data Storage

The data storage for the virtual vehicle is defined in some shared global variables in the ovms.h module. These variables are as follows:

Vehicle Identification

The vehicle identification parameters are concerned with the identification of the vehicle and the environment of the OVMS hardware in the vehicle.

- unsigned char ovms_firmware[3]

  Stores the version of the OVMS firmware. This parameter is set by the ovms.c module itself, during initialization. It may be read by vehicles, but is typically not written. The parameter itself is 3 decimal bytes (e.g.; 0x02,0x01,0x01 denotes version v2.1.1).

- unsigned char car_type[5]

  Stores the vehicle type currently identified, as a 1 to 4 character, zero-terminated, ASCII string. This is typically set during initialization of the vehicle plug-in module, and often updated to better reflect the specific vehicle model based on CAN bus messages.

- unsigned char car_vin[18]

Stores the vehicle VIN (normally detected on the CAN bus).

- unsigned int car_12vline

  The power of the 12 volt system in the car. This is expressed in units of 0.1 volts (so a value of 121 denotes 12.1volts). This parameter is normally maintained by the 'inputs' module, based on using an ADC reading the 12V power line to the module, but vehicle plug-in modules can update it if they have a more accurate method.

- unsigned char car_gsmcops[9]

  The GSM provider currently being used, as a zero-terminated string of between 1 and 8 characters. This value is usually maintain by the 'net' module directly.

GPS Status

The GPS status parameters are concerned with the physical location of the vehicle, and are usually determined by a GPS unit.

The OVMS hardware module has a built-in GPS unit that can be enabled on demand, or the GPS location can be read from a GPS unit in the vehicle itself. The determination of which to use is set by global variable 'net_fnbits'. If bit0 of net_fnbits is unset, then the OVMS GPS unit will not be used, and it will be up to the car module to provide GPS location (usually from values read from the CAN bus). If bit0 of net_fnbits is set, then the OVMS GPS unit will be controlled by the 'net' module and these GPS status values updated automatically, without the plug-in vehicle module having to do anything.

Usually, the plug-in vehicle module makes the decision as to whether to enable OVMS GPS based on the capabilities of the vehicle itself.

Here are the parameters themselves:

- unsigned char car_gpslock

  This is set to 0 if there is no good GPS lock, or 1 if the GPS is providing location data.

- signed char car_stale_gps

  This is set to -1 if there is no GPS data, 0 if there is data, but it is stale (ie;

perhaps shows outdated information), and >0 if GPS location data is correct and up-to-date. Vehicle modules may use this as a timer to detect stale data; so it is common to see it count down from high numbers – so long as it is >0, then the GPS data should be considered valid.

- signed long car_latitude

  The latitude of the vehicle, as a signed long integer (4 bytes). To convert a floating decimal latitude, multiply by 2048*3600.

- signed long car_longitude

  The longitude of the vehicle, as a signed long integer (4 bytes). To convert a floating decimal longitude, multiply by 2048*3600.

- unsigned int car_direction

  The direction of the vehicle, to the nearest degree between 0 and 359.

- signed int car_altitude

  The altitude of the vehicle, specified in integer metres.

Tire Pressure Monitoring System

The status of the vehicle's tire pressure monitoring system is stored in several parameters:

- signed char car_stale_tpms

  An indication whether the TPMS values are present, or stale. A value -1 indicates no TPMS present, 0 is TPMS present but values stale, and >0 is TPMS present and values are correct and up-to-date. Vehicle modules may use this as a timer to detect stale data; so it is common to see it count down from high numbers – so long as it is >0, then the TPMS data should be considered valid.

- signed char car_tpms_t[4]

  TPMS tire temperatures. Values are in celcius, offset by 40 (so, for example, 10 celcius would be stored as 50). The array elements are for

the four wheels front-right [0], rear-right [1], front-left [2] and rear-left [3].

- unsigned char car_tpms_p[4]

  TPMS pressures. Values are in PSI multiplied by 0.2755. The array elements are for the four wheels front-right [0], rear-right [1], front-left [2] and rear-left [3].

Driving Status

The driving status of the vehicle is stored in the following parameters:

- unsigned char car_doors1 [bit 7]

  A bit used to signal whether the car is turned on or off. Set to 1 if the car ignition switch is ON, otherwise 0.

- unsigned char car_speed

  The vehicle speed, in vehicle units (either mile-per-hour, or kilometers-per-hour, depending on the module configuration).

- unsigned int car_trip

  The vehicle trip counter, defined in units of 0.1 miles. Note that the interpretation of this parameter is currently under discussion, and may change – we anticipate changing the units to vehicle units at some point in the future.

- unsigned long car_odometer

  The vehicle odometer, defined in units of 0.1 miles. Note that the interpretation of this parameter is currently under discussion, and may change – we anticipate changing the units to vehicle units at some point in the future.

- unsigned char car_drivemode

  The current drive mode, i.e. selected engine profile. This has no common specification, can be stored in vehicle-specific encoding.

- unsigned int car_power

  Current power level at the battery, defined in units of 0.1 kW. Use positive figures for discharging, negative for charging.

- unsigned long car_energy_used

  Energy used on the current trip so far, defined in units of 1 Wh.

- unsigned long car_energy_recd

  Energy recovered on the current trip so far, defined in units of 1 Wh. This may be used for charging energy count as well.

Vehicle Environment

The temperatures in and around the vehicle are stored in the following parameters:

- signed char car_stale_ambient

  An indication whether the ambient temperature value is valid or stale. A value -1 indicates no ambient temperature can be measured, 0 is ambient temperature is measured but value is stale, and >0 is ambient temperature is measured and value is correct and up-to-date. Vehicle modules may use this as a timer to detect stale data; so it is common to see it count down from high numbers – so long as it is >0, then the ambient temperature should be considered valid.

- signed char car_ambient_temp

  Ambient temperature, in degres celcius.

- signed char car_stale_temps

  An indication whether the three internal temperatures are valid or stale. A value -1 indicates no internal temperatures can be measured, 0 is internal temperatures are measured but values are stale, and >0 is internal temperatures are measured and values are correct and up-to-date. Vehicle modules may use this as a timer to detect stale data; so it is common to see it count down from high numbers – so long as it is >0, then

the three internal temperatures should be considered valid.

- signed char car_tpem

  The temperature of the PEM (electronics controlling the power entering / leaving the battery), specified in degrees celcius.

- unsigned char car_tmotor

  The temperature of the MOTOR, specified in degrees celcius.

- signed int car_tbattery

  The temperature of the BATTERY, specified in degrees celcius.

The environment of the vehicle itself is stored in the following parameters:

- unsigned char car_doors1 [bit 0]

  This bit is set to 1 if the front left door is OPEN, else 0.

- unsigned char car_doors1 [bit 1]

  This bit is set to 1 if the front right door is OPEN, else 0.

- unsigned char car_doors2 [bit 6]

  This bit is set to 1 if the bonnet is OPEN, else 0.

- unsigned char car_doors2 [bit 7]

  This bit is set to 1 if the trunk is OPEN, else 0.

- unsigned char car_doors1 [bit 2]

  This bit is set to 1 if the charge port is OPEN, else 0.

- unsigned char car_doors1 [bit 6]

  This bit is set to 1 if the vehicle handrake is ON, else 0.

- unsigned char car_doors2 [bit 5]

This bit is set to 1 if the vehicle headlights or ON, else 0.

- unsigned char car_doors3 [bit 1]

  This bit is set to 1 to indicate the vehicle is awake, and operational, else 0. For some cars, this would indicate that cooling systems are working, but for most cars it should just be set to 1 if the car is ‘awake’ in any way.

- unsigned char car_doors2 [bit 4]

  This bit is set to 1 if the car supports a valet mode and that mode is ENABLED, else 0.

- unsigned char car_doors2 [bit 3]

  This bit is set to 1 if the car is LOCKED, else 0.

- unsigned char car_lockstate

  Set to 4 if vehicle is locked, or 5 if unlocked. Note that this parameter is deprecated and will be removed in future versions of the OVMS virtual vehicle. In preference, car_doors [bit 3] should be used to determine lock state of the vehicle.

- unsigned char car_doors4 [bit 2]

  This bit is set to 1 if the car alarm is SOUNDING, else 0.

The virtual vehicle includes a parking timer, to indicate whether the vehicle is parked, and for how long:

- unsigned long car_time

  Time (in seconds), as measured by the car. The 0 value of this can be whatever the car module desires. It is acceptable to merely initialize this to zero and increment it by 1 for every ticker1() received. Alternatively, a more accurate method is to read the real time clock from the vehicle CAN bus (if available) and update from there.

- unsigned long car_parktime

  The time (in seconds) that the vehicle was parked, or 0 if not parked.

Battery Status

The status of the battery is stored in the following parameters:

- unsigned char car_SOC

  The state of charge of the vehicle battery (expressed as a percentage, with 0 denoting empty and 100 denoting full). Note that most vehicle batteries are not linear, and there is considerable variance and guesswork in calculating capacity – this is, at best, a guess.

- unsigned int car_idealrange

  The vehicle ideal range, specified in miles. Note that this can be calculated by taking the manufacturer's published estimate for range (over available SOC) and multiplying by car_SOC/100. Note that the interpretation of this parameter is currently under discussion, and may change – we anticipate changing the units to vehicle units at some point in the future.

- unsigned int car_estrange

  The vehicle estimated range, specified in miles. Driving patterns are not consistent, and most calculations of estimated range are performed by using past driving habits (which may or may not be an indication of future driving habits). This is, at best, a guess. Note that the interpretation of this parameter is currently under discussion, and may change – we anticipate changing the units to vehicle units at some point in the future.

Charging Status

The charging status is monitored by a large number of parameters:

- unsigned char car_doors1 [bit 3]

  This bit is set to 1 if the pilot signal is present, else 0. This would normally indicated that the vehicle is connected to external power and either charging or ready to charge.

- unsigned char car_doors1 [bit 4]

This bit is set to 1 if the vehicle is currently charging, else 0.

- unsigned char car_linevoltage

  This indicates the line voltage (in units of 1 volt) that the vehicle is currently charging at. If not currently charging then the value should be ignored as irrelevant (and probably set to 0 by the vehicle module).

- unsigned char car_chargecurrent

  This indicates the line current (in units of 1 amp) that the vehicle is currently charging at. If not currently charging then the value should be ignored as irrelevant (and probably set to 0 by the vehicle module).

- unsigned char car_chargemode

  This value indicates the charge/drive mode that the vehicle is currently in. It is one of 0 (standard), 1 (storage), 3 (range) or 4 (performance). For vehicles that don’t support such modes, it should be set to 0.

- unsigned char car_charge_b4

  This value is currently not used, and should be set to 0 and ignored by Apps.

- unsigned char car_chargestate

  The current charging state (or result of last charge). It is one of 1 (charging), 2 (top off), 4 (done), 13 (preparing to charge), or 21-24 (stopped charging).

- unsigned char car_chargesubstate

  The current charging sub-state (or result of last charge). It is one of 2 (scheduled start), 3 (by request), 5 (timerwait), 7 (connect power cable), 14 (interrupted).

- unsigned char car_chargelimit

  The charge current limit (in units of 1 amp) currently in effect. This is used by some vehicles to limit the charge current available.

- unsigned int car_chargeduration

  The charge duration (in minutes) for the current / just completed charge.

- unsigned int car_chargekwh

  The charge kWh put into the battery for the current / just completed charge.

- unsigned int car_cac100

  An indication of battery health, storing the Calculated Amp Hour capacity (or some other suitable metric) of the battery in units of 0.01Amp. For example, a battery with a CAC of 160Ah would be stored as 16000.

- unsigned char car_soh

  State of battery health in percent (0-100). This can be calculated from the cac100 (or vice versa) or read from the battery management system. The BMS may have another concept of "battery health" than just capacity.

- signed int car_chargefull_minsremaining

  During charging, and if supported, the number of minutes of charging remaining until the battery is 100% full (for the current charge mode). Or, the value -1 if not supported or car not charging.

- signed int car_chargelimit_minsremaining

  During charging, and if supported, the number of minutes of charging remaining until the charge reaches a predefined limit. Or, the value -1 if not supported or car not charging.

- unsigned int car_chargelimit_rangelimit

  The range limit (in vehicle units) for car_chargelimit_minsremaining. Or, the value -1 if no range limit defined.

- unsigned char car_chargelimit_soclimit

  The SOC% limit for car_chargelimit_minsremaining. Or, the value -1 if no SOC% limit defined.

Some vehicles may support a delayed timer charge mode. We have started to support this in OVMS, as these parameters:

- signed char car_stale_timer

  An indication whether the charging timer is valid or stale. A value -1 indicates no charging timer is supported, 0 is charging timer is working but value is stale, and >0 is charging timer is supported and value is correct and up-to-date. Vehicle modules may use this as a timer to detect stale data; so it is common to see it count down from high numbers – so long as it is >0, then the charging timer should be considered valid.

- signed char car_timermode

  Set to 0 for charge on-plugin, or 1 for timer-delayed.

- unsigned int car_timerstart

  The time that charging is scheduled to start. This is specified as the number of minutes past midnight UTC less 1. For example, 59 is 1am GMT, 1439 (is midnight GMT).

Vehicle Module Development Checklists

Development Checklists

You can use these checklists to know what vehicle parameters the OVMS system supports, and how you can map a specific vehicle to these.

Parameter	Purpose	Vehicle Support Notes
	Vehicle type identified Vehicle VIN	

Parameter	Purpose	Vehicle Support Notes
	Does the vehicle have a built-in GPS? If so, complete the following. Set to record GPS lock Denotes GPS data staleness Latitude of the vehicle Longitude of the vehicle Direction of the vehicle Altitude of the vehicle	

Parameter	Purpose	Vehicle Support Notes
	Does the vehicle have a TPMS? If so, complete the following. Denotes TPMS data staleness The temperatures of each wheel The pressures of each wheel	

Parameter	Purpose	Vehicle Support Notes
	Set if the car ignition is ON The speed of the vehicle The vehicle trip counter The vehicle odometer	

Parameter	Purpose	Vehicle Support Notes

Parameter	Purpose	Vehicle Support Notes
	Ambient temperature staleness	
	Ambient temperature	
	Other Temperature staleness	
	Temperature of the PEM	
	Temperature of the MOTOR	
	Temperature of the BATTERY	
	Set if front left door OPEN	
	Set if front right door OPEN	
	Set if bonnet is OPEN	
	Set if trunk is OPEN	
	Set if charge port is OPEN	
	Set if handbrake is ON	
	Set if headlights ON	
	Set if vehicle is AWAKE (or if not supported just follow ignition)	
	Set if valet mode is ENABLED	
	Set if vehicle is LOCKED	
	4=locked, 5=unlocked (deprecated)	
	Set if alarm SOUNDING	
	Time (in seconds)	
	Time (seconds) vehicle parked	

Parameter	Purpose	Vehicle Support Notes
	State of charge percentage	
	Ideal range	
	Estimated range	
	Calculated Amp Hour capacity	

Parameter	Purpose	Vehicle Support Notes
	Set if pilot signal ON	
	Set if vehicle CHARGING	
	Line voltage while charging	
	Line current while charging	
	Vehicle Mode	
	Charging state	
	Charging sub-state	
	Charge current limit	
	Charge duration (minutes)	
	Charge kWh put into battery	
	Mins remaining for full charge	
	Mins remaining for limited charge	
	Desired range for limited charge	
	Desired SOC% for limited charge	
	Charge timer staleness	

Parameter	Purpose	Vehicle Support Notes
	Charge timer mode	
	Charge timer schedule	

Vehicle-Specific Extensions

NET_MSG Command Handlers

A vehicle module that requires to override or implement a new NET_MSG command can hook in to *vehicle_fn_commandhandler*.

BOOL (*vehicle_fn_commandhandler)
(BOOL msgmode, int code, char* msg)

Once implemented, the hook will be called whenever a NET_MSG message is received by the module.

- *msgmode* is TRUE if the GPRS system is currently sending responses that can be added to by the vehicle module, or FALSE otherwise.
- *code* is the message request code.
- *msg* is the message request body.

If the vehicle module handles the message request, it should return TRUE. Otherwise, it should return FALSE to allow the NET_MSG system to handle it.

NET_SMS SMS Handlers

A vehicle module that requires to override or implement a new NET_SMS SMS command can hook in to *vehicle_fn_smshandler*.

BOOL (*vehicle_fn_smshandler)
(BOOL premsg, char *caller, char *command, char *arguments)

Once implemented, the hook will be called whenever a NET_SMS message is received by the module.

- *premsg* is TRUE if the hook is called at the start of SMS message processing (before any other default SMS handling).
- *caller* is the caller telephone number.
- *command* is the SMS command.
- *arguments* are the arguments to the SMS command.

If the vehicle module handles the SMS command it should return TRUE. Otherwise, it should return FALSE to allow the NET_SMS system to handle it.

NET_SMS SMS Extensions

A vehicle module that requires to supplement a standard SMS response to a NET_SMS SMS command can hook in to *vehicle_fn_smsextensions*.

BOOL (*vehicle_fn_smsextensions)
(char *caller, char *command, char *arguments)

Once implemented, the hook will be called whenever a NET_SMS message is handled by the module, after the standard response.

- *caller* is the caller telephone number.
- *command* is the SMS command.
- *arguments* are the arguments to the SMS command.

If the vehicle module extends the SMS command it should return TRUE. Otherwise, it should return FALSE to allow the NET_SMS system to handle it.

OVMS Server Development

OVMS App Development

Conclusions