OVMS

Open Vehicle Monitoring System

www.openvehicles.com

OVMS Developer Guide
v3.0.2 (24th August 2022)

For OVMS Hardware Module v3.x and Firmware v3.x.x

History

v3.0.0	20th Sep 2017	Initial version written
v3.0.1	20th Sep 2020	New chapter: Developing with Visual Studio Code
v3.0.2	24th Aug 2022	Minor edit to title of tools section for Linux/Mac

Table of Contents

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.

A short history of OVMS

The Open Vehicle Monitoring System (OVMS) started out as a hobbyist project to bring cellphone control to the Tesla Roadster electric car. Three hobbyist, Michael Stegen, Mark Webb-Johnson, and Sonny Chen, built OVMS from the ground-up as an open source project.

The first batch of in-vehicle modules (now called 'v1') were hand-built in time for christmas 2011. These modules combined a vehicle control board (based on the PIC18F2680 processor) and a standard SIMCOM SIM900 cellular base board, housed in a relatively large white box.

In the early days, developers and those extremely technically capable were the only ones using OVMS, but as time went by more and more users wanted to benefit from cellular monitoring of their electric vehicles, and a community was born.

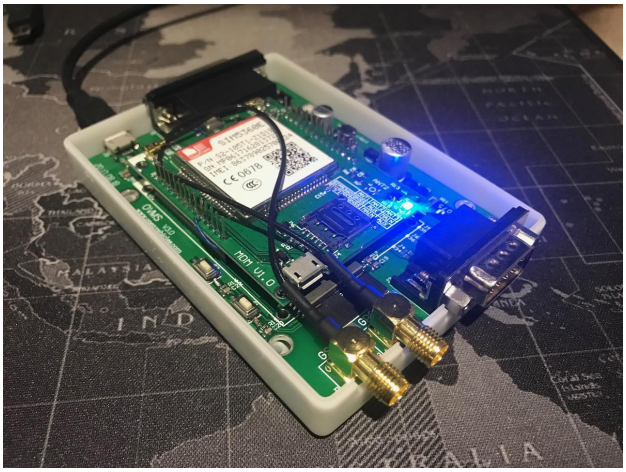
As the number of users grew, and the pain of hand-producing circuit boards got worse with volume, the developers started work on a new version of the hardware (now called 'v2'). This would use a standard metal industrial enclosure, a single mass-produced PCB (with cellular or

one side, and PIC18F2685 processor on the other), and most importantly an industry standard DB9 connector allow vehicles other than the Tesla Roadster to be supported via vehicle-specific adaptor cables. We also upgraded the SIMCOM to a SIM908 (and later SIM808) module, which supported GPS (for those vehicle types without onboard GPS). This new version was released late in 2012, with Fasttech (in China) as the distributor.

Both the v1 and v2 hardware modules used the same firmware code, and 2G SIMCOM modems. Over the years, the limitation of this hardware become more and more restrictive, as developers continued to push the limits with more vehicle support, and more advanced functionality. By 2016, a dozen vehicle types were supported, and both RAM and FLASH memory were at capacity. In addition, the phase-out of 2G by AT&T in USA began to be an issue.

So, in 2017, the core development team began work on a version 3 (now called ‘v3’) of the OVMS module. This would be a complete re-write of the module firmware on a new platform:

- Espressif IDF
  (based on Open Source GNU C/C++)
- ESP32 dual-core ARM processor
- 16MB flash and 520KB RAM
- Wifi
- Bluetooth
- 3x CAN buses
- GPIO expansion ports
- A new DB26 external expansion port
- An internal expansion slot
- SD CARD support
- USB port
- 3G and 4G modem options



The first ten v3 developer modules were produced in September 2017.

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.

Developing with Visual Studio Code (on Windows)

This chapter describes how you install and successfully build and flash OVMS using Visual Studio Code on Windows with the current (2020-07-14) version of OVMS. It may be applicable to other OSes as well.

This will be a portable installation. Meaning you can do everything on a flash drive and can take it to any other computer. You cannot (easily) just change the location on a computer though as some software saves their absolute path in their config. Nevertheless the whole installation won't save anything in AppData/User/etc. folders. So a backup of the installation folder will backup everything.
In this example everything will be installed in D:\OVMS which will be the root path and the `user-name` will be `soko` (change it to your Windows user name). The only important thing is that the whole path needs to be ASCII characters only with no spaces, symlinks or accents.

ESP-IDF (toolchain)

(Based on:
https://docs.espressif.com/projects/esp-idf/en/release-v3.3/get-started/windows-setup.html)

The current version of the OVMS specific ESP-IDF (https://github.com/openvehicles/esp-idf) is v3.3. So download the toolchain v3.3:
https://dl.espressif.com/dl/esp32_win32_msys2_environment_and_toolchain_idf3-20200601.zip

Open the ZIP-file and extract the content of its containing msys32 folder to `C:\OVMS` so the path to mingw32.exe is `C:\OVMS\mingw32.exe`.

Using a higher toolchain/xtensa version

It can happen that the xtensa version in the zip file downloaded above is too old (happened 2020-07-16). The newer version should be lister at https://docs.espressif.com/projects/esp-idf/en/release-v3.3/get-started/windows-setup-scratch.html#updating-existing-windows-environment under section "Alternative Setup: Just download a toolchain". Download this zip file. Then goto `C:\OVMS\opt` , delete the `xtensa-esp32-elf` folder that's in there and replace it with the one in the zip file.

Install OVMS's own ESP-IDF

Doubleclick on this `mingw32.exe` (don't use mingw64.exe!). This is your environment bash shell.
Type in and execute:

```
$ git clone https://github.com/openvehicles/esp-idf.git
$ cd esp-idf
```

```
$ git submodule update --init --recursive
$ cd ..
```

Now the OVMS specific version of ESP-IDF is on your computer. To later pull an ESP IDF update, issue:

```
$ cd esp-idf
$ git pull
$ git submodule update --recursive
$ cd ..
```

Now tell the toolchain the setup path of ESP-IDF by creating a new file named `export_idf_path.sh` in `C:\OVMS\etc\profile.d` with the content

```
export IDF_PATH="C:/OVMS/home/soko/esp-idf"
```

(use forward slashes and don't forget to replace `soko` by your user name!).

Close the window and start `mingw32.exe` again. Type

```
$ printenv IDF_PATH
```

Which should give you the path you put into the file if everything is correct.

Next install the required python packages. To do so type

```
$ python -m pip install --user -r $IDF_PATH/requirements.txt
```

Get and build the OVMS source code

Type the following to get the latest version of OVMS from github:

```
$ git clone https://github.com/openvehicles/Open-Vehicle-Monitoring-System-3.git
$ cd Open-Vehicle-Monitoring-System-3/vehicle/OVMS.V3
$ git submodule update --init --recursive
```

Next set the default build configurations:

```
$ cp support/sdkconfig.default.hw31 sdkconfig
$ make menuconfig
```

The last command will open a DOS-GUI window. Just choose "EXIT" for now which will close the window. After a while the prompt will come back.

If everything worked out as it should the source should build successfully by typing:

```
$ make -j17 all
```

The `17` behind `-j` should be the number of your CPU cores in your machine + 1 (so 17 for 16 cores). This option just speeds up the build process.

Congrats! You have now your own build OVMS firmware in the file

```
C:\OVMS\home\soko\Open-Vehicle-Monitoring-System-3\vehicle\OVMS.V3\build\ovms3.bin
```

You can now put this file onto a HTTP-server/website and use the firmware update option “OTA” on the OVMS web interface. Unfortunately there is no way to upload this file directly to OVMS.

Optional: Flashing the OVMS hardware directly

As an option you can connect the OVMS box directly via USB to your computer and flash it. This may be dangerous though as a broken firmware may render your hardware inaccessible.

Install USB driver for the OVMS

So Windows recognize the OVMS as a virtual COM/Serial port a driver for the “CP210x USB to UART Bridge” is needed from here:
https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

Download the correct version for your Windows version and unpack it somewhere. Plug in the OVMS via USB and point Windows to the unpacked folder when it asks for the driver.

Now you need to know the serial port name (i.e. “COM3”) the OVMS was installed as. Open the Windows Device Manager to find out.

Tell ESP-IDF the serial port

Start `mingw32.exe` again and type:

```
$ cd Open-Vehicle-Monitoring-System-3/vehicle/OVMS.V3
$ make menuconfig
```

Don’t close the DOS-GUI window. Instead choose “Serial flasher config” then the first entry “...Default serial port”. Delete whatever is written there, type `COM3` and press Enter. Select EXIT two times and answer YES when asked if you want to save the changes.

Flash the OVMS

To build and flash, instead of just building type:

```
$ cp make -j17 flash
```

Install and configure portable Visual Studio Code and Git

Programming with a decent IDE with Code Completion makes the day to day development so much easier. Here is how to use Visual Studio Code (VSC) and Git as portable installation. Until now you used the build in Git of ESP-IDF. VSC can’t use this unfortunately. This is why you have to install it again.

Install VSC portable

Download current version as a ZIP-file from https://code.visualstudio.com/download
Unpack the content of the ZIP-file to `C:\OVMS\VSC` and then create the folder `C:\OVMS\VSC\data` which will make the installation a portable installation.

Install Git for Windows portable

Download the current version as portable EXE from https://git-scm.com/download/win
Use the EXE-File itself to unpack the content to `C:\OVMS\GIT`
Hint: The EXE-unpacking executes a batch file at the end which ties this installation to this absolute path. So it cannot be moved anymore afterwards.

Configure VSC

Now it's time to start VSC the first time with `C:\OVMS\VSC\Code.exe`. This usually triggers the installation of the C/C++ IntelliSense extension. Just let it happen… It might also show up later when opening the OVMS folder.
Tell VSC the path of the portable Git installation by opening "File", "Preferences", "Settings". Then "Extensions", "Git" and scroll until you find the "Path" section. Click on "Edit in settings.json". In file edit this line to:

```
   "git.path": "C:\\OVMS\\GIT\\cmd\\git.exe"
```

And save it. Close VSC for now.

To tell VSC how to cooperate with the OVMS Source you need some additional files. In the Windows Explorer create this folder (be aware it has a dot in front. So it's ".vscode". Depending on your Explorer settings it might not allow you to create such a folder or removes the dot automatically)
`c:\OVMS\home\soko\Open-Vehicle-Monitoring-System-3\vehicle\OVMS.V3\.vscode\`

Place the following three files with the listed content into this folder:
`c_cpp_properties.json:`

```
   {
       "configurations": [
           {
               "name": "OVMS",
               "includePath": [
                   "${workspaceFolder}",
               "${workspaceFolder}\\..\\..\\..\\esp-idf\\components\\**",
               "${workspaceFolder}\\**"
               ],
               "defines": [
                   "_DEBUG",
                   "UNICODE",
```

```
                "_UNICODE"
            ],
            "intelliSenseMode": "msvc-x64"
        }
    ],
    "version": 4
}
```

settings.json:

```
{
    "terminal.integrated.shell.windows": "C:\\OVMS\\usr\\bin\\bash.exe",
    "terminal.integrated.shellArgs.windows": ["--login"],
    "terminal.integrated.env.windows": {
        "MSYSTEM": "MINGW32",
        "CHERE_INVOKING":"1"
    },
    "files.associations": {
        "forward_list": "cpp"
    }
}
```

tasks.json (again change the -j17 to your number):

```
{
    "version": "2.0.0",
    "tasks": [
      {
        "type": "shell",
        "label": "ALL",
        "command": "make -j17 all",
        "options": {
          "cwd": "${workspaceFolder}"
        },
        "problemMatcher": [
          "$gcc"
        ],
        "group": {
          "kind": "build",
          "isDefault": true
        }
      },
      {
        "type": "shell",
        "label": "FLASH",
        "command": "make -j17 flash",
        "options": {
          "cwd": "${workspaceFolder}"
        },
```

```
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    },
    {
      "type": "shell",
      "label": "CLEAN",
      "command": "make -j17 clean",
      "options": {
        "cwd": "${workspaceFolder}"
      },
      "problemMatcher": [
        "$gcc"
      ],
      "group": {
        "kind": "build",
        "isDefault": true
      }
    }
  ]
}
```

Coding OVMS in VSC

Start VSC again and choose “File”, “Open Folder…” and choose the folder

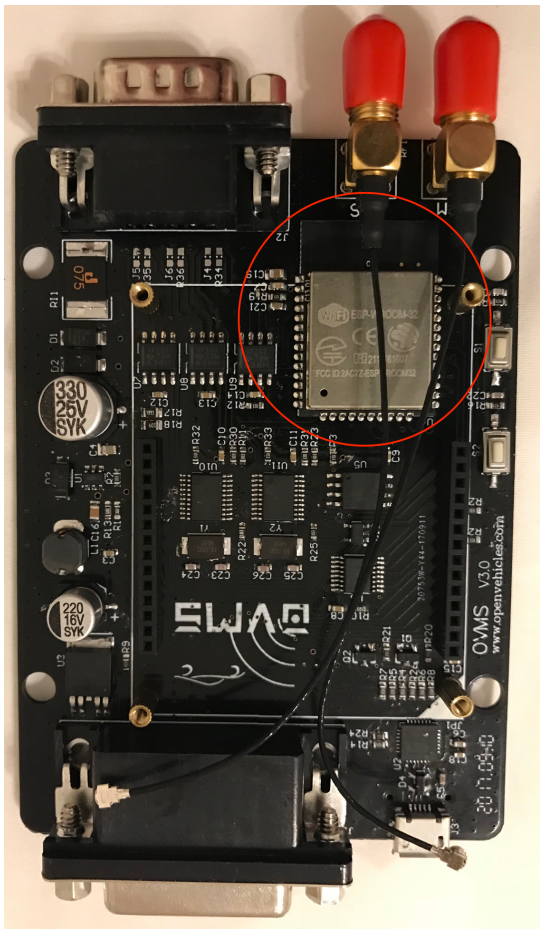
```
C:\OVMS\home\soko\Open-Vehicle-Monitoring-System-3\vehicle\OVMS.V3
```

Now choose “Terminal”, “New Terminal” and press the “Accept”-Button on the windows that pop up. Open a New Terminal again. Now you should see the same shell as with `mingw32.exe`.

To build inside VSC just select “Terminal”, “Run Task” and then “ALL” which executes the “make -j17 all” command of the tasks.json file.

Have fun coding in VSC with IntelliSense and using it’s integrated Git features! It’s not as good as “Visual Studio 2019” but better than nothing ;)

Vehicle Firmware Development Tools (Linux/Mac)



Version 3 (v3) of the Open Vehicle Monitor System module is based on an Espressif ESP-32 microcontroller module (Wifi and Bluetooth support, dual core ARM processor, RAM and FLASH).

The Software Development Kit used is Espressif's ESP IDF, which is based on the open source GCC compiler plus a large number of support libraries and FreeRTOS based operating system. ESP IDF runs under Windows, Linux and Mac OSX.

You need to use the OVMS fork of the Espressif ESP IDF. It is a direct clone of the Espressif IDF, but includes some modifications especially for OVMS. You can find that here::

https://github.com/openvehicles/esp-idf

and you can find setup guides at the same place.

The usual installation steps are:

1. Install pre-requisite tools
2. Install xtensa toolchain
3. Git clone the ESP IDF repository
4. Setup the environment

The latest ESP IDF instructions are here, (but don't use the latest version of the esp-idf install instructions, because the OVMS fork is a couple of versions older) :

Use the esp-idf instructions for the release version matching the current OVMS branch version (currently: 3.3)

https://docs.espressif.com/projects/esp-idf/en/release-v3.3/get-started/index.html

selected from the esp-idf-readthedocs.io documentation, to ensure that you get the correct versions of the prerequisites and toolchain.

After installing the prerequisites and the xtensa toolchain, Git clone the OVMS ESP IDF repository:

```
$ git clone https://github.com/openvehicles/esp-idf.git
```

```
$ cd esp-idf
$ git submodule update --init --recursive
```

To later pull an ESP IDF update, issue:

```
$ cd esp-idf
$ git pull
$ git submodule update --recursive
```

Once you've got the Espressif ESP IDF installed, we recommend you try to compile one or more examples to make sure your environment works as expected.

Git clone the OVMS v3 repository and init the build configuration:

```
$ git clone https://github.com/openvehicles/Open-Vehicle-Monitoring-System-3.git
$ cd Open-Vehicle-Monitoring-System-3/vehicle/OVMS.V3
$ git submodule update --init --recursive
$ cd vehicle/OVMS.V3
$ cp support/sdkconfig.default.hw31 sdkconfig
$ make menuconfig
```

The config tool may prompt for some new config options, use the default values for all. Check the flasher serial port, leave other options at defaults.

Build and flash your first OVMS firmware to a connected V3 module:

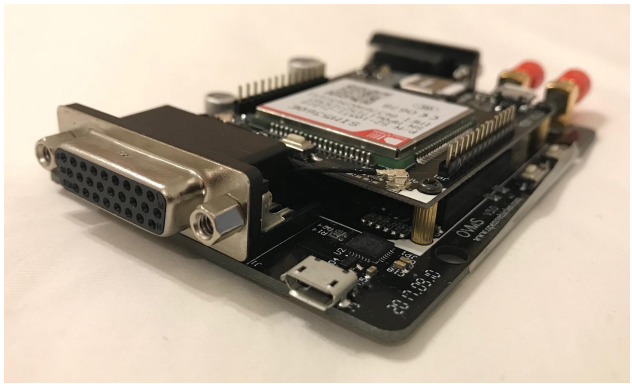
```
$ make flash
```

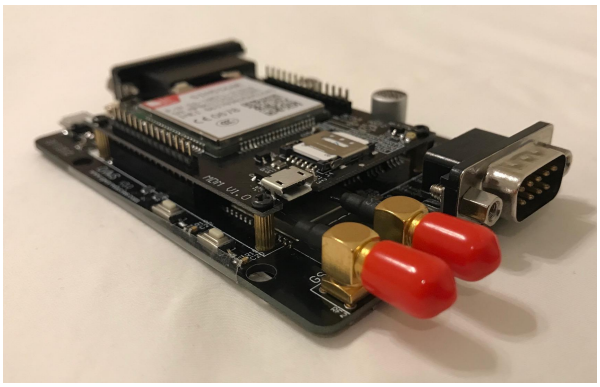
Note to Mac OS X users with *brew*: If you have upgraded your OS, to High Sierra for instance, and you've already had tools installed via *brew* before: You will need to upgrade some component(s) such as bison as Apple has deeply modified some underlying libraries, otherwise you might end up having *make* commands fail. Launching the following command will download and install the adapted build even if it is the same version.

```
$ brew upgrade bison
```

This has been sufficient (with the proper paths) to get things working properly.

Module USB Development Port





The base OVMS module has a micro USB port next to the big DB26 expansion connector. This is the port connected to the ESP-32 microcontroller. It uses a Silicon Labs CP2102 USB-to-UART controller. If your operating system doesn't have this driver already installed, you can download and install it from the Silicon Labs website.

https://www.silabs.com/products/development-tools/software/usb-to-uart-bridge-vcp-drivers

Linux note: if your distribution includes the braille display driver "brltty", you may need to uninstall that, as it claims any CP2102 device to be a braille device. This applies e.g. to openSuSE 15.0.

There is a second USB port on the optional modem board (above the GSM and GPS antennas, and next to the SIM card slot). This provides direct access to the USB on the SIMCOM modem module, and requires special SIMCOM drivers. It is used for firmware updates, and direct debugging of the SIMCOM module, so can generally be ignored.

Note that OVMS v3 is currently in prototype stage, and has not received CE/FCC approval. In particular, while the v3.0 circuit board has protection diodes on the power supply pins, it will most likely require extra ESD and line noise filtering before approval. It is hobbyist kit.

As such, we recommend against directly connecting to the USB ports on your computer. Instead, please use a powered USB hub to ensure that the module receives adequate power, and there is some isolation between the module and your computer.

The module can be powered by USB (including the optional modem), and provides a serial console as well as flash programming capability. You can use Espressif's 'make monitor' tool to connect to the module as a serial terminal.

```
$ make monitor
rst:0x1 (POWERON_RESET),boot:0x1f (SPI_FAST_FLASH_BOOT)
...
I (45) boot: ESP-IDF v2.1-2-g7138fb02 2nd stage bootloader
I (46) boot: compile time 09:50:02
I (79) boot: Enabling RNG early entropy source...
I (79) boot: SPI Speed      : 40MHz
I (79) boot: SPI Mode       : DIO
I (87) boot: SPI Flash Size : 16MB
I (100) boot: Partition Table:
I (111) boot: ## Label            Usage          Type ST Offset   Length
I (134) boot:  0 nvs              WiFi data        01 02 00009000 00004000
I (157) boot:  1 otadata          OTA data         01 00 0000d000 00002000
I (180) boot:  2 phy_init         RF data          01 01 0000f000 00001000
I (204) boot:  3 factory          factory app      00 00 00010000 00400000
I (227) boot:  4 ota_0            OTA app          00 10 00410000 00400000
I (250) boot:  5 ota_1            OTA app          00 11 00810000 00400000
I (273) boot:  6 store            Unknown data     01 81 00c10000 00100000
...
I (1755) heap_alloc_caps: Initializing. RAM available for dynamic allocation:
I (1778) heap_alloc_caps: At 3FFAFF10 len 000000F0 (0 KiB): DRAM
I (1798) heap_alloc_caps: At 3FFC9D20 len 000162E0 (88 KiB): DRAM
I (1819) heap_alloc_caps: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM
I (1840) heap_alloc_caps: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (1862) heap_alloc_caps: At 40094FD0 len 0000B030 (44 KiB): IRAM
...
I (1883) cpu_start: Pro cpu start user code
I (1940) command: Initialising COMMAND (1000)
I (1952) events: Initialising EVENTS (1200)
I (1954) config: Initialising CONFIG (1400)
...
I (1556) ovms_main: Starting HOUSEKEEPING...
I (1556) housekeeping: Initialising HOUSEKEEPING Framework...
I (1616) housekeeping: Executing on CPU core 1
I (1616) housekeeping: Starting PERIPHERALS...
I (1616) peripherals: Initialising OVMS Peripherals...
I (1626) peripherals:   ESP32 system
I (1626) peripherals:   SPI bus
I (1626) peripherals:   MAX7317 I/O Expander
I (1636) peripherals:   ESP32 CAN
I (1636) peripherals:   ESP32 WIFI
I (1646) peripherals:   ESP32 BLUETOOTH
I (1646) peripherals:   ESP32 ADC
I (1656) peripherals:   MCP2515 CAN 1/2
I (1656) peripherals:   MCP2515 CAN 2/2
I (1666) peripherals:   SD CARD
I (1666) peripherals:   SIMCOM MODEM
I (1666) housekeeping: Starting USB console...

Welcome to the Open Vehicle Monitoring System (OVMS) - Async Console
OVMS >
```

There are some important things shown in that boot screen, so let's look in some detail.

```
$ make monitor
rst:0x1 (POWERON_RESET),boot:0x1f (SPI_FAST_FLASH_BOOT)
```

The OVMS v3 module includes two push-button switches on the motherboard, BOOT and ENABLE. Pressing the ENABLE switch will reset the processor. Holding down the BOOT switch, while pressing ENABLE, will boot into firmware download mode.

To simplify development, and flashing of modules, OVMS includes some transistor logic controlled by async control lines (RTS, DTR, etc) to control this ENABLE and BOOT functionality. When you type "make flash", Espressif will connect to the serial port, then manipulate the control lines to hold down BOOT and then toggle ENABLE, to switch to firmware download mode. By default, every time you type "make monitor", Espressif will also toggle the ENABLE pin to reset the device. You can change this behaviour in "make menuconfig", or use your own terminal emulator, to avoid this (if you don't want it).

```
I (45) boot: ESP-IDF v2.1-2-g7138fb02 2nd stage bootloader
I (46) boot: compile time 09:50:02
I (79) boot: Enabling RNG early entropy source...
I (79) boot: SPI Speed      : 40MHz
I (79) boot: SPI Mode       : DIO
I (87) boot: SPI Flash Size : 16MB
```

The default flash in the ESP-32 WROOM-32 module we use is 4MB, but that is insufficient. So, OVMS includes an external 16MB flash and the ESP-32 chips we use are modified to permanently burn fuses inside the chip, in order to select and use this external flash. At the moment, we are using 40Mhz DIO mode to access this chip; but we may switch to QIO mode and increase the speed later when we are comfortable with the performance requirements and stability.

```
I (100) boot: Partition Table:
I (111) boot: ## Label            Usage          Type ST Offset   Length
I (134) boot:  0 nvs              WiFi data        01 02 00009000 00004000
I (157) boot:  1 otadata          OTA data         01 00 0000d000 00002000
I (180) boot:  2 phy_init         RF data          01 01 0000f000 00001000
I (204) boot:  3 factory          factory app      00 00 00010000 00400000
I (227) boot:  4 ota_0            OTA app          00 10 00410000 00400000
I (250) boot:  5 ota_1            OTA app          00 11 00810000 00400000
I (273) boot:  6 store            Unknown data     01 81 00c10000 00100000
```

By default, OVMS partitions this 16MB flash to NVS (general parameter=value key storage), otadata (control data for the Over-The-Air update facility), phy_init (RF data), factory (the firmware App flashed at the factory), ota_1 (OTA image #1), ota_2 (OTA image #2), and store (a FAT filesystem mounted as /store during boot).

The factory, ota_0 and ota_1 partitions are all 4MB in size - and that is the largest firmware image that can be loaded onto the chip. The otadata partition contains a pointer to one of these three partitions, to tell the bootloader which to map to during boot.

When first delivered, we will be running the factory firmware in the factory partition, and otadata will point to that. If an Over-The-Air update is performed, it will be downloaded to ota_0 and otadata modified to point to ota_0. Upon next reboot, the bootloader will then run the image in ota_0. If a second Over-The-Air update is performed, while running in ota_0 partition, it will be flashed to ota_1, and then otadata modified to point to ota_1. Note that all the operating addressing of all these images is the same, irrespective of the partition offset in the flash - the bootloader deals with mapping physical memory to these partitions, via it's cache. If the bootloader fails to boot into an OTA partition, it will fall back to the factory partition. In this way, the factory partition is always there as a backup.

Note that 'make flash' will flash all these partitions (including bootloader, partition table, factory partition, etc). Generally, it is quicker to 'make app-flash' to flash just the factory application partition.

```
I (1755) heap_alloc_caps: Initializing. RAM available for dynamic allocation:
I (1778) heap_alloc_caps: At 3FFAFF10 len 000000F0 (0 KiB): DRAM
I (1798) heap_alloc_caps: At 3FFC9D20 len 000162E0 (88 KiB): DRAM
I (1819) heap_alloc_caps: At 3FFE0440 len 00003BC0 (14 KiB): D/IRAM
I (1840) heap_alloc_caps: At 3FFE4350 len 0001BCB0 (111 KiB): D/IRAM
I (1862) heap_alloc_caps: At 40094FD0 len 0000B030 (44 KiB): IRAM
```

Memory is limited. This is an embedded system. Take care.

```
I (1883) cpu_start: Pro cpu start user code
I (1940) command: Initialising COMMAND (1000)
I (1952) events: Initialising EVENTS (1200)
I (1954) config: Initialising CONFIG (1400)
```

OVMS boots in a very defined order. If you need a component to hook into this initialisation system, it is generally best to contact the lead developers for information on where precisely to hook in.

```
I (1556) ovms_main: Starting HOUSEKEEPING...
I (1556) housekeeping: Initialising HOUSEKEEPING Framework...
I (1616) housekeeping: Executing on CPU core 1
I (1616) housekeeping: Starting PERIPHERALS...
I (1616) peripherals: Initialising OVMS Peripherals...
I (1626) peripherals:   ESP32 system
I (1626) peripherals:   SPI bus
I (1626) peripherals:   MAX7317 I/O Expander
I (1636) peripherals:   ESP32 CAN
I (1636) peripherals:   ESP32 WIFI
I (1646) peripherals:   ESP32 BLUETOOTH
I (1646) peripherals:   ESP32 ADC
```

```
I (1656) peripherals:   MCP2515 CAN 1/2
I (1656) peripherals:   MCP2515 CAN 2/2
I (1666) peripherals:   SD CARD
I (1666) peripherals:   SIMCOM MODEM
I (1666) housekeeping: Starting USB console...

Welcome to the Open Vehicle Monitoring System (OVMS) - Async Console
OVMS >
```

Once the boot is complete, you'll be presented with the OVMS async console prompt, and can type commands.

ESP-32 WROOM-32 Module eFuses

As part of the factory production process for OVMS v3, we custom burn some ESP-32 eFuses on the WROOM-32 module to suite our requirements (in particular for external 16MB flash). If you ever replace the WROOM-32 module, or wish to use custom hardware with OVMS firmware and flash partition layout, you will need to custom-burn these eFuses. Below is the documentation for what we do.

```
espefuse.py -p <path-to-uart> burn_efuse XPD_SDIO_REG 1
espefuse.py -p <path-to-uart> burn_efuse XPD_SDIO_TIEH 1
espefuse.py -p <path-to-uart> burn_efuse XPD_SDIO_FORCE 1
espefuse.py -p <path-to-uart> burn_efuse SPI_PAD_CONFIG_CLK 6
espefuse.py -p <path-to-uart> burn_efuse SPI_PAD_CONFIG_Q 7
espefuse.py -p <path-to-uart> burn_efuse SPI_PAD_CONFIG_D 8
espefuse.py -p <path-to-uart> burn_efuse SPI_PAD_CONFIG_HD 9
espefuse.py -p <path-to-uart> burn_efuse SPI_PAD_CONFIG_CS0 22
```

If you do it right, it will look like this:

```
Config fuses:
XPD_SDIO_FORCE          Ignore MTDI pin (GPIO12) for VDD_SDIO on reset    = 1 R/W (0x1)
XPD_SDIO_REG            If XPD_SDIO_FORCE, enable VDD_SDIO reg on reset   = 1 R/W (0x1)
XPD_SDIO_TIEH           If XPD_SDIO_FORCE & XPD_SDIO_REG, 1=3.3V 0=1.8V   = 1 R/W (0x1)
SPI_PAD_CONFIG_CLK      Override SD_CLK pad (GPIO6/SPICLK)                = 6 R/W (0x6)
SPI_PAD_CONFIG_Q        Override SD_DATA_0 pad (GPIO7/SPIQ)               = 7 R/W (0x7)
SPI_PAD_CONFIG_D        Override SD_DATA_1 pad (GPIO8/SPID)               = 8 R/W (0x8)
SPI_PAD_CONFIG_HD       Override SD_DATA_2 pad (GPIO9/SPIHD)              = 9 R/W (0x9)
SPI_PAD_CONFIG_CS0        Override SD_CMD pad (GPIO11/SPICS0)                   = 22 R/W
(0x16)
DISABLE_SDIO_HOST       Disable SDIO host                                 = 0 R/W (0x0)
```

Note that this is a one-time operation. Once these eFuses are burned, the change is permanent and irreversible.

Vehicle Firmware Overview

Command Line Interpreter

NOTE: This documentation is no longer maintained. Please see https://docs.openvehicles.com/en/latest/cli/index.html for the current documentation.

The command line interpreter or command parser presented by the OVMS async serial console is constructed as a tree of command word tokens. Enter a single "?" followed by <RETURN> to get the root command list, like this:

```
OVMS# ?
.               Run a script
bms             BMS framework
boot            BOOT framework
can             CAN framework
...
```

A root command may be followed by one of a list of additional tokens called subcommands which in turn may be followed by further subcommands down multiple levels. The command and subcommand tokens may be followed by parameters. Use "?" at any level in the command sequence to get the list of subcommands applicable at that point. If the next item to be entered is a parameter rather than a subcommand, then a usage message will be displayed to indicate the required or optional parameters. The usage message will also be shown if the command as entered is not valid. The usage message is described in further detail below.

Command tokens can be abbreviated so long as enough characters are entered to uniquely identify the command, then optionally pressing <TAB> will auto-complete the token. If the abbreviated form is not sufficient to be unique (in particular if no characters have been entered yet) then <TAB> will show a concise list of the possible subcommands and then retype the portion of the command line already entered so it can be completed. Pressing <TAB> is legal at any point in the command; if there is nothing more that can be completed automatically then there will just be no response to the <TAB>.

The OvmsCommand::RegisterCommand() function is used to add command and subcommand tokens are added to the tree. New commands are added to the root of the tree using the global MyCommandApp.RegiserCommand(). Subcommands are added as children of a command using the OvmsCommand pointer returned when RegisterCommand() is called for the parent, thus building the tree. For example:

```
OvmsCommand* cmd_wifi = MyCommandApp.RegisterCommand("wifi","WIFI framework",
```

wifi_status);
cmd_wifi->RegisterCommand("status","Show wifi status",wifi_status);
cmd_wifi->RegisterCommand("reconnect","Reconnect wifi client",wifi_reconnect);

The RegisterCommand() function takes the following arguments:

- const char* **name** – the command token
- const char* **title** – one-line description for command list
- void (***execute**)(...) – does the work of the command
- const char ***usage** – the "Usage:" line describing parameters
- int **min** – minimum number of parameters allowed
- int **max** – maximum number of parameters allowed
- bool **secure** – true for commands permitted only after "enable"
- int (***validate**)(...) – validates parameters as explained later

It's important to note that many of these arguments can and should be defaulted. The default values are as follows:

- **execute** = NULL
- **usage** = ""
- **min** = 0
- **max** = 0
- **secure** = true
- **validate** = NULL

For example, for secure, non-terminal commands, such as the top-level "framework" commands like "bms" in the list of root commands shown earlier, the model should simply be:

```
RegisterCommand("name", "Title");
```

For secure, terminal (sub)commands that don't require any additional parameters, the model should be:

```
RegisterCommand("name", "Title", execute);
```

This model also applies if the command has children but the command itself wants to execute a default operation if no subcommand is specified. It is incorrect to specify **min** = 0, **max** = 1 to indicate an optional subcommand; that is indicated by the presence of the **execute** function at the same time as a non-empty children array.

Any command with required or optional parameters should provide a **usage** string hinting about the parameters in addition to specifying the minimum and maximum number of parameters allowed:

RegisterCommand("name", "Title", execute, "usage", min, max);

The **usage** argument only needs to describe the parameters that follow this (sub)command because the full usage message is dynamically generated. The message begins with the text "Usage: " followed by names of the ancestors of the this (sub)command back to the root of the tree plus the name of this (sub)command itself. That is, the message starts with all the tokens entered to this point followed by additional description of subcommands or parameters that may be entered next, as determined by the **usage** string.

The **usage** string syntax conventions for specifying alternative and optional parameters are similar to those of usage messages in Unix-like systems. The string can also include special codes to direct the dynamic generation of the message:

- $C expands to the list of children commands as child1|child2|child3.
- [$C] expands to optional children as [child1|child2|child3].
- G expands to the usage string of the first child; this would typically used after $C so the usage message shows the list of children and then the parameters or next-level subcommands that can follow the children. This is useful when the "usage" string is the same for all or most of the children as in this example:

  Usage: power adc|can1|can2|can3|egpio|esp32|sdcard|simcom|spi|wifi deepsleep|devel|off|on|sleep|status

- $Gfoo$ expands to the usage of the child named "foo"; this variant would be used when not all the children have the same usage but it would still be helpful to show the usage of one that's not first.
- $L lists a separate full usage message for each of the children. This provides more help than just showing the list of children but at the expense of longer output.
- For terminal (sub)commands (those with no children) that take additional parameters, the **usage** string contains explicit text to list the parameters:
  - Parameter names or descriptions are enclosed in angle brackets to distinguish the them from command tokens, for example "<metric> <value>".
  - Parameters that are optional are further enclosed in square brackets, like "<id> <name> [<value>]".
  - When there are alternative forms or meanings for a parameter, the alternatives are separated by vertical bar as in "[<task names or ids>|*|=]" which indicates that the parameter can be either of the characters '*' or '=' instead of a list of task names or ids. An variant form encloses the alternatives in curly braces as in "<param> {<instance> | *}".
  - One or more additional lines of explanatory text can be included like this: "<id>\nUse ID from connection list / 0 to close all".

For non-terminal commands (those with children subcommands) the **usage** argument can be omitted because the default value of "" is interpreted as "$C". For commands that have children subcommands that are optional (because an **execute** function is included) the default **usage** argument is interpreted as “[$C]”.

Most commands do not need to specify a **validate** function. It supports two extensions of the original command parser design:

1. For commands that store the possible values of a parameter in a NameMap<T> or CNameMap<T>, the **validate** function enables <TAB> auto-completion when entering that parameter. For example, see the "config" command in main/ovms_config.cpp.
2. The original design only allowed parameters to be collected by the terminal subcommand. That forced an unnatural word order for some commands. The **validate** function enables non-terminal subcommands to take one or more parameters followed by multiple children subcommands. The parameters may allow <TAB> completion if the possible values are stored in NameMap<T> or CNameMap<T> or they could be something else like a number that can be validated by value. The **validate** function must indicate success for parsing to continue to the children subcommands. The return value is the number of parameters validated if successful or -1 if not.

The "location" command is an example that includes an intermediate parameter and also utilizes the "$L" form of the usage string:

```
OVMS# location action enter ?
Usage: location action enter <location> acc <profile>
Usage: location action enter <location> homelink 1|2|3
Usage: location action enter <location> notify <text>
```

See components/ovms_location/src/ovms_location.cpp for the implementation of the location_validate() function and the RegisterCommand() calls to build the command subtree.

OVMS Vehicle Modules

All functionality to support a particular type of vehicle should be implemented in OVMS vehicle modules. These are in the following directories:

components/vehicle_*

Steps to take, to stub support for a new vehicle type, are:

1. Decide on a **short type code** for the vehicle (upper case, minimum two characters), that doesn't conflict with any existing vehicle (`vehicle list` shows the currently defined vehicles). Build the code from an abbreviation of the brand/manufacturer and an abbreviation of the vehicle. Examples:
    a. `TR` – Tesla Roadster
    b. `RT` – Renault Twizy
    c. `VWUP` – VW e-Up

2. Decide on a **long vehicle name**. This is normally just the vehicle name itself (such as "Tesla Roadster").

3. Decide on a **unique log tag** to use in all your modules. Use prefix "`v-`" followed by a lower case short name of the vehicle for the log tag. The logging system allows to set different log levels per log tag. See existing vehicle modules for reference on how to define and use the log tag. Examples:
    a. "`v-twizy`"
    b. "`v-vweup`"
    c. "`v-bmwi3`"

4. Decide on a **unique namespace prefix** for your custom configuration, metrics and commands. The custom namespace shall be a lower case code beginning with "`x`" followed by at least two characters for the vehicle type. Try to keep this at three characters in length. Examples:
    a. "`xrt`" – Renault Twizy
    b. "`xvu`" – VW e-Up
    c. "`xi3`" – BMW i3/i3s

5. Create the directory structure

   components/vehicle_<lower-case-long-name>
   /component.mk
   /src
   /vehicle_<lower-case-long-name>.{h,cpp}

   You can use vehicle_none as a template for this.

6. Change main/Kconfig to add a menu configuration item in the “Vehicle Support” section

   OVMS_VEHICLE_<upper-case-long-name>

7. Change your component.mk to selectively compile in support for your vehicle, based on that CONFIG_OVMS_VEHICLE_* menu configuration.

8. Add all the above to git, make sure it compiles, commit and push.

Once basic support is stubbed, as above, you can start work on actual implementation of the vehicle module. We suggest you look at vehicle_teslaroadster for a practical example of how things should be done. These notes may help.

- A short OvmsVehicleXXXInit object handles registration of your vehicle name, with associated vehicle implementation object, at init_priority 9000. All vehicle initialise at this priority level, unless they are dependent on other vehicle modules.

- A reception queue, and task, will be created and managed for you automatically.

- Create a member function “void IncomingFrameCan1(CAN_frame_t* p_frame)” in your vehicle implementation object. This will be used to receive incoming CAN bus messages on CAN bus #1. You can also create IncomingFrameCan2 and IncomingFrameCan3 for the other two buses (if you require). The vehicle framework will automatically deliver CAN bus messages to your functions.

- Implement the constructor for your vehicle implementation. You will need to define the CAN buses your vehicle requires in the vehicle constructor function, with RegisterCanBus(int bus, CAN_mode_t mode, CAN_speed_t speed). Here is a simple example:

```
OvmsVehicleTeslaRoadster::OvmsVehicleTeslaRoadster()
  {
  ESP_LOGI(TAG, "Tesla Roadster v1.x, v2.x and v3.0 vehicle module");
  RegisterCanBus(1,CAN_MODE_ACTIVE,CAN_SPEED_1000KBPS);
```

```
}
```

- Implement a destructor for your vehicle implementation to perform any necessary cleanup. The base vehicle module destructor will automatically handle cleanup of the task, queue, and any CAN buses you are using.

- Implement your IncomingFrameCanX handlers to process the incoming CAN frames. You will usually be decoding the messages, and updating metrics with the decoded vehicle data, appropriately.

You can use the '`vehicle set …`' console command to switch to your vehicle.

Note that vehicle specific command registration can also be made in the OvmsVehicleXXXInit constructor. This is the preferred mechanism for implementing vehicle-specific commands. You can use MyVehicleFactory.m_currentvehicle to make sure a vehicle is currently set to run, and VehicleName() on that, to ensure it is your vehicle type.

Vehicle Metrics

OVMS v3 is all about ‘metrics’. Vehicle modules update the metrics, and apps consume them via network protocols. A metric is a single piece of information, and its associated current value.

The metrics subsystem is implemented in main/ovms_metrics.{h,cpp} as a C++ object. The primary interface to metrics is via the MyMetrics static object. Using that interface, new metrics can be registered, and values set. In addition, listeners can be established to be notified whenever a particular metric changes.

The metrics themselves are implemented in a virtual base class OvmsMetric. Currently, OVMS supports metrics of types boolean (OvmsMetricBool), integer (OvmsMetricInt), floating point (OvmsMetricFloat) and string (OvmsMetricString). All metric types support the base functionality on OvmsMetric for setting and reading the current value as strings (converting as necessary to the base data type), as well as indicators for whether an initial value has been defined, and for tracking modifications.

A global METRICS_MAX_MODIFIERS is used to define the maximum number of Modifiers (services tracking modifications to metrics). This is implemented internally as a bitmask. Every time a metric is modified, all the bits are set true. Then, individual modifiers can track their own individual bit, and clear as necessary. Modifiers themselves register to receive their own unique modifier bit number with MyMetrics.RegisterModifier(). In this way, modifiers can choose to use their modifier bit as they see fit. For example, the server protocols register as modifiers and clear their modifier bit when they send the data to the server; in this way, they can reduce the data transmissions by only sending modified metrics.

Standardised metrics are automatically registered on system boot by the code in main/metrics_standard.{h,cpp}. In addition, a static object StandardMetrics is defined, with member variables pointing to each of the standard metrics. This allows fast, efficient, type safe, access to the standardised metrics. For example:

```
StandardMetrics.m_v_bat_soc->SetValue(42);
```

Vehicle-specific metrics can be registered and maintained by individual vehicle modules. In most cases, a vehicle module should maintain a pointer to the registered metric (using a member variable of the vehicle object) for fast and efficient access. The registration itself is automatic (all that is necessary is to create a metric object from one of the OvmsMetric derived metric types, and they will be registered automatically, by name). In most cases, the vehicle module initialisation should use MyMetrics.Find(name) to search for a metric by name and store the resulting pointer. If not found, the metric should be first-time registered.

There are some console commands available to view and manipulate the metrics (for diagnostic and testing purposes):

```
OVMS > metrics list
m.freeram                     46716
m.hardware                    OVMS WIFI BLE BT cores=2 rev=ESP32/1
m.serial                      aa:bb:cc:dd:ee:ff
m.tasks                       18
m.version                     3.0.0/factory/main build Sep 22 2017 06:33:43
s.v2.connected                yes
s.v2.peers                    0
v.b.12v                       0

OVMS > metrics set <metric> <value>
```

Metrics have several common attributes that deserve further explanation:

- Last Modified Date: This is accessible via LastModified(). Note that this date is the time the metric was last updated. If a metric is updated, but the value is unchanged, the last modified date will be updated, but the modified bit flags will not.

- Stale: This is accessible via IsStale(), and setable via SetStale(bool stale). Whenever a metric is updated (irrespective of whether the value has been changed or not), the stale flag is cleared.

- AutoStale: This is configurable when the metric object is created, or with the SetAutoStale(int seconds) function. If set (non zero), the stale flag will be automatically set if the metric value has not been updated for the specified number of seconds.

Configuration

The configuration for OVMS v3 is stored on ESP32 flash, under vfs mount /store/ovms_config. Under normal configuration, that is a protected path and the normal VFS operations (cp, cat, etc) will not be permitted. As a developer, you can disable that protection via a menuconfig option (Component Config -> OVMS -> Developer Options -> “Disable the usual protections for configuration visibility in VFS”).

The general design of the configuration store is that configuration items are identified by:

```
Parameter [instance] = value
```

The parameters must be registered, usually during boot-time initialisation:

```
RegisterParam(
  std::string name,
  std::string title,
  bool writeable=true,
  bool readable=true);
```

The ‘writable’ and ‘readable’ flags control whether the user has write and/or read access to all the instances and values within that parameter. Firmware modules themselves always have full read/write access, and these only control user console access permissions. For example, you might set a store for passwords as writable=true, readable=false, so users could set passwords, but not view them. Similarly, a parameter maintained by a firmware module (and not the user) might be writable=false, readable=true.

If you don’t require an instance, then simply use ‘*’ or ‘’ (empty string).

The configuration system provides ‘config list’ and ‘config set’ commands to maintain these configuration values.

Configuration values themselves are always stored internally as strings. Convenience functions are provided to access these as String, Int, Float and Bool.

Access config flash filesystem

You can read and write the flash content using the esptool.py, and you can mount the retrieved image using a loopback device to access the config store on your PC.

Example: to retrieve and mount the FAT filesystem using losetup:

```
$ cat partitions.csv
# OVMS 16MB flash ESP32 Partition Table
# Name,   Type, SubType, Offset,   Size
nvs,      data, nvs,     0x9000,  0x4000
otadata,  data, ota,     0xd000,  0x2000
phy_init, data, phy,     0xf000,  0x1000
factory,  app,  factory, 0x10000,  4M
ota_0,    app,  ota_0,   ,         4M
ota_1,    app,  ota_1,   ,         4M
store,    data, fat,     ,         1M

$ bc
65536+(3*4*1024*1024)
==> 12648448
1024*1024
==> 1048576

$ ~/esp/esp-idf/components/esptool_py/esptool/esptool.py --port /dev/tty.SLAB_USBtoUART
--baud 921600 read_flash 12648448 1048576 fatty
esptool.py v2.2.1
…
1048576 (100 %)
Read 1048576 bytes at 0xc10000 in 12.9 seconds (649.4 kbit/s)…

$ file fatty
fatty: DOS/MBR boot sector, code offset 0xfe+2, OEM-ID "MSDOS5.0", Bytes/sector 4096, FAT
1, root entries 512, sectors 250 (volumes <=32 MB) , Media descriptor 0xf8, sectors/FAT 1,
sectors/track 63, heads 255, serial number 0x210000, unlabeled, FAT (1Y bit by descriptor)

$ mkdir f
$ losetup /dev/loop0 fatty
$ mount -t vfat /dev/loop0 f
$ ls -l f
drwxr-xr-x 0 root root 4096 Jan  1  1980 ovms_config

$ umount f
$ losetup -d /dev/loop0
```

On macOS, use hdiutil instead of losetup:

```
$ hdiutil attach -imagekey diskimage-class-CRawDiskImage -nomount fatty
/dev/disk11
$ mount_msdos /dev/disk11 f
$ ls -l f
drwxr-xr-x 0 root root 4096 Jan  1  1980 ovms_config
```

```
$ umount f
$ hdiutil detach /dev/disk11
“Disk11” ejected.
```

See `esptool.py --help` on writing back a modified image.
To wipe the entire flash, and re-flash with latest:

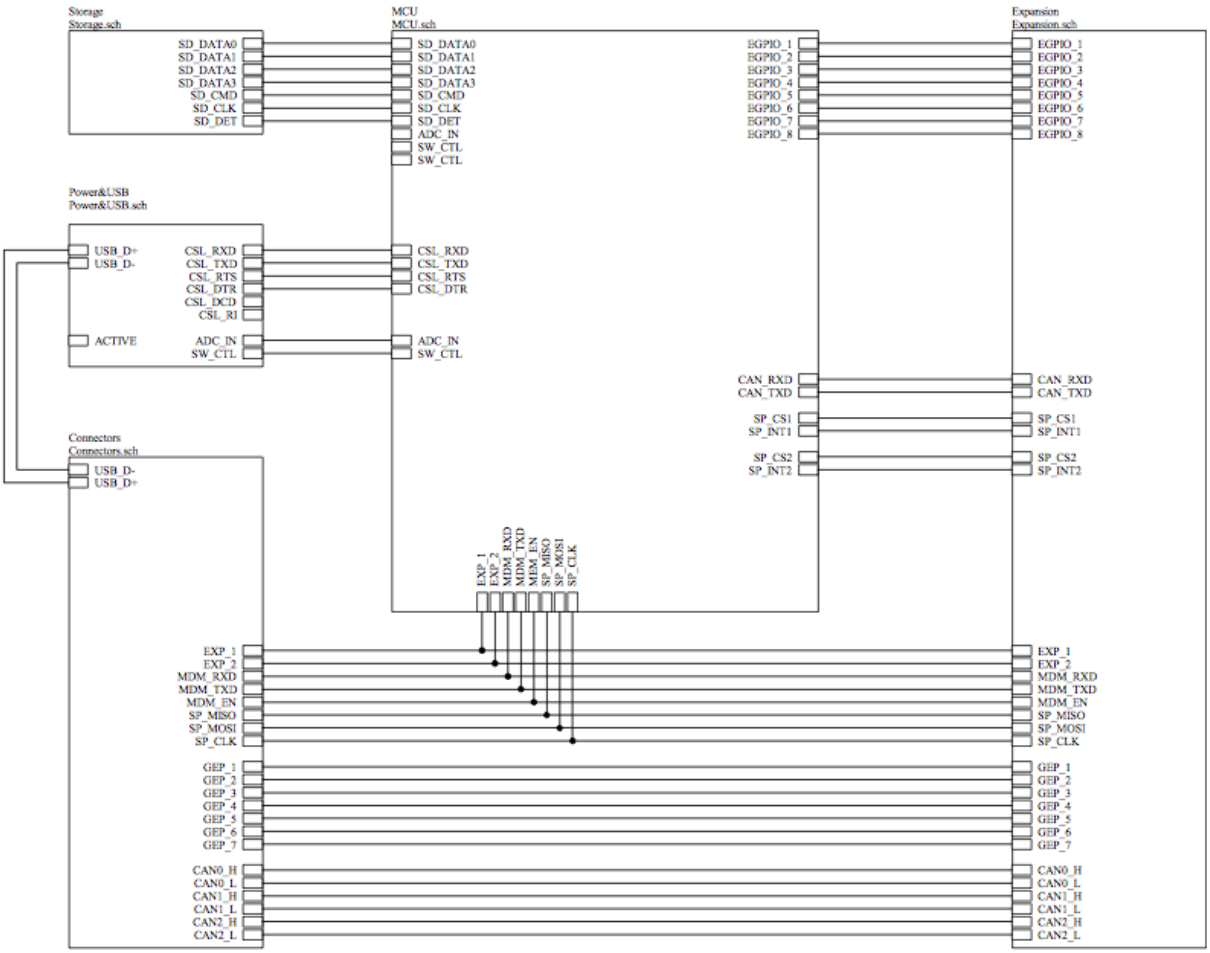
```
$ make erase_flash
$ make flash
```

You could probably just erase that one flash partition, but the above is quick and simple.

Module Circuit Design

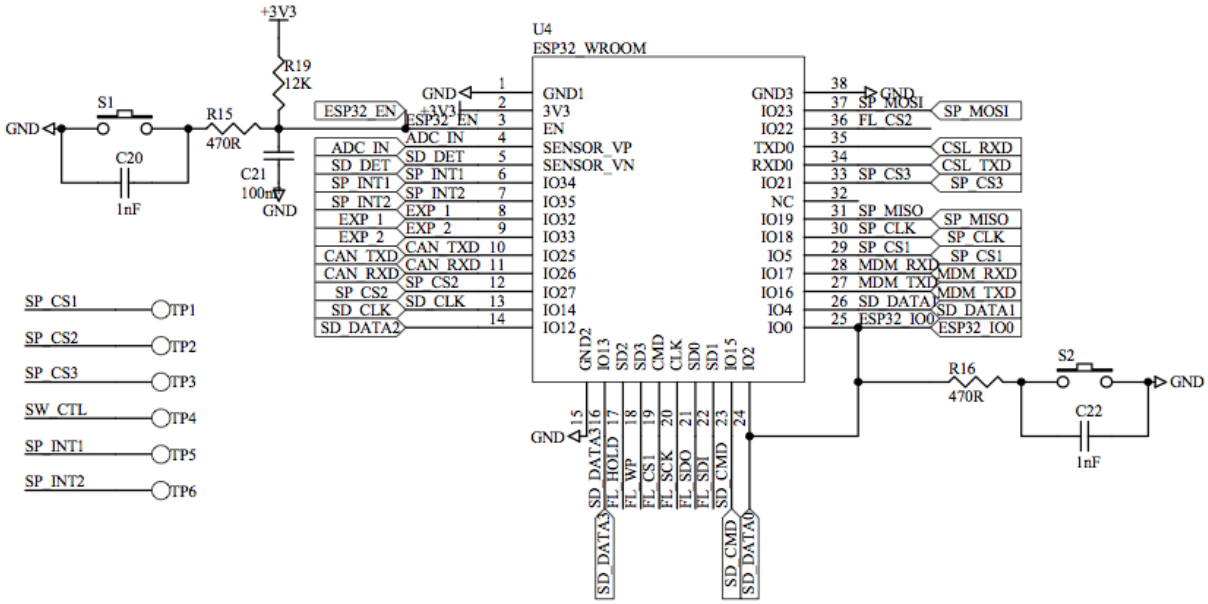
The OVMS v3 module circuit is designed to be extremely expandable, and to serve as the base for vehicle CAN bus hacking.

Main Board Overview

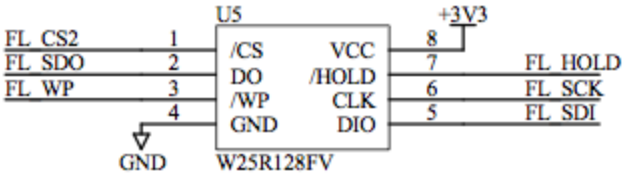


At the centre of the OVMS main board is an ESP32 WROOM-32 module. From there, we have modules for Storage, Power & USB, and Expansion Connectors.

WROOM-32 and Support Circuits

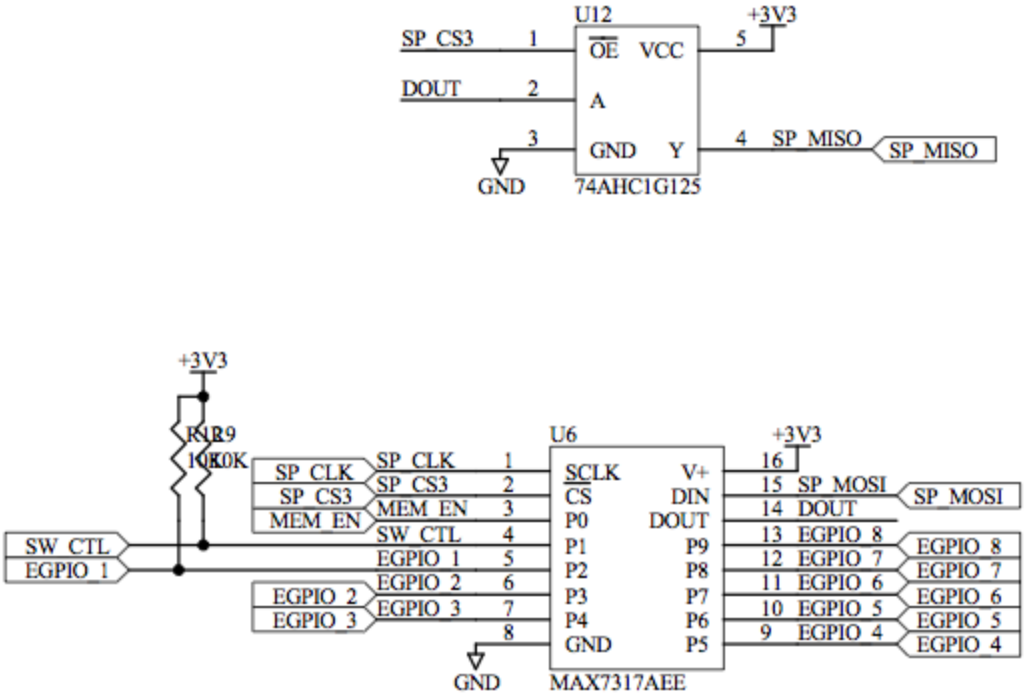


128MB External SPI Flash



The on-board 4MB SPI flash of the ESP32 WROOM-32 module is not used by OVMS v3. Instead, an external W32R128FV 128MB SPI flash is used for bootloader, firmware, and configuration storage. This is connected to the same SPI bus lines as the onboard 4MB flash, but uses a FL_CS2 (IO22) as the SPI chip select.

MAX7317 GPIO Expansion



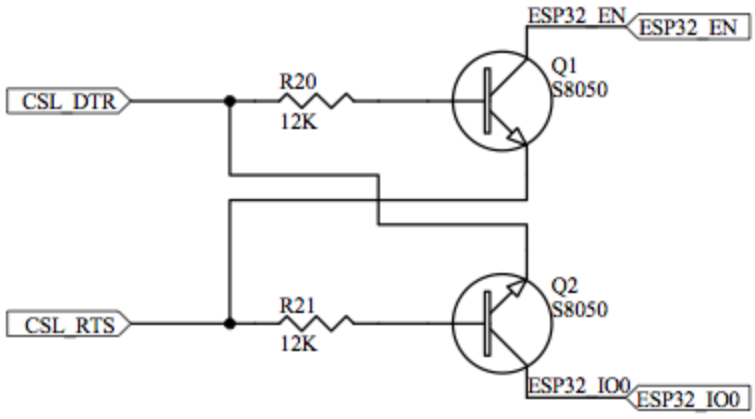
To expand on the number of GPIO available, a MAX7317 GPIO expansion chip is used, and driven by the 2nd ESP32 SPI bus. This shares the same bus as the two MCP2515 CAN bus controllers, and has chip select on CP_CS3 (IO21). To allow this chip to co-exist with others on that shared SPI bus, a 74AHCIG125 tri-state buffer is used to isolate the SP_MISO line (driven from the same SP_CS3 line as the MAX7317 itself).

The MAX7317 provides 10 EGPIO pins (labelled P0 through P9), which can be configured as either input or output pins. The first few of these are reserved, and used by other modules in the OVMS v3 system.

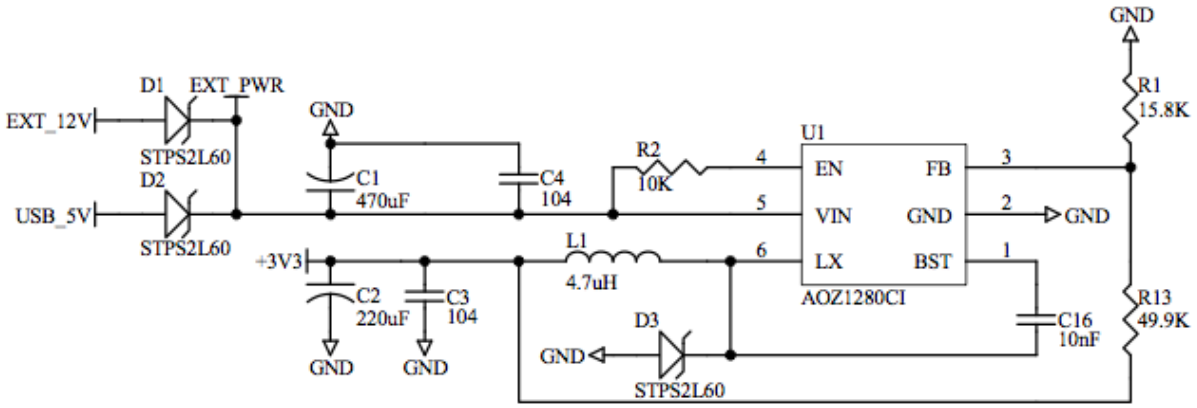
MAX7317 Pin Assignments		
MAX7317 EGPIO	**Pin**	**Comment**
P0	MDM_EN	Modem Enable Line (for expansion modem control)
P1	SW_CTL	Switched 12V control
P2	ESP32CAN_EN	ESP32 CAN SN65 Transceiver power control
P3	MDM_DTR	Modem DTR Line (for expansion modem control)
P4	EGPIO3	General purpose GPIO (available for expansion)
P5	EGPIO4	General purpose GPIO (available for expansion)
P6	EGPIO5	General purpose GPIO (available for expansion)
P7	EGPIO6	General purpose GPIO (available for expansion)
P8	EGPIO7	General purpose GPIO (available for expansion)
P9	EGPIO8	General purpose GPIO (available for expansion)

ESP32 Boot/Flash Control via USB

The DTR and RTS pins from CP2102 USB control lines are used in the normal ESP32 fashion to control bootloader boot/flash mode, using transceiver logic. These pins control the ESP32_EN and ESP32_IO0 pins during boot time (also controllable via the two push button switches on the circuit board).



AOZ1280CI Power

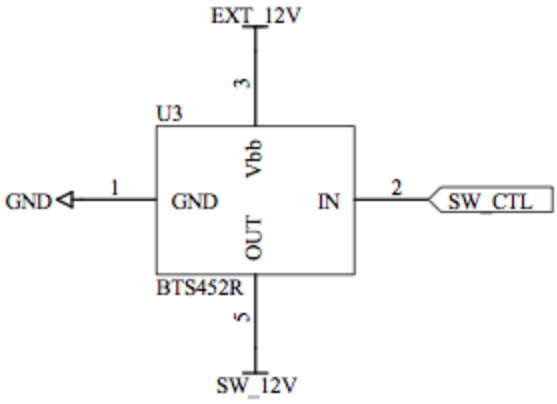


An AOZ1280CI switched mode power supply chip is used to convert either USB 5V or external 12V incoming power to regulated 3.3V. Diodes are used for polarity protection, as well as to correctly protect the USB 5V side from external 12V power. A shared ground scheme is used.

BTS452R Switched 12V Power

A BTS452R, connected to SW_CTL on the MAX7317 GPIO expansion chip, is used to provide switched 12V power on the DA26 expansion connector.

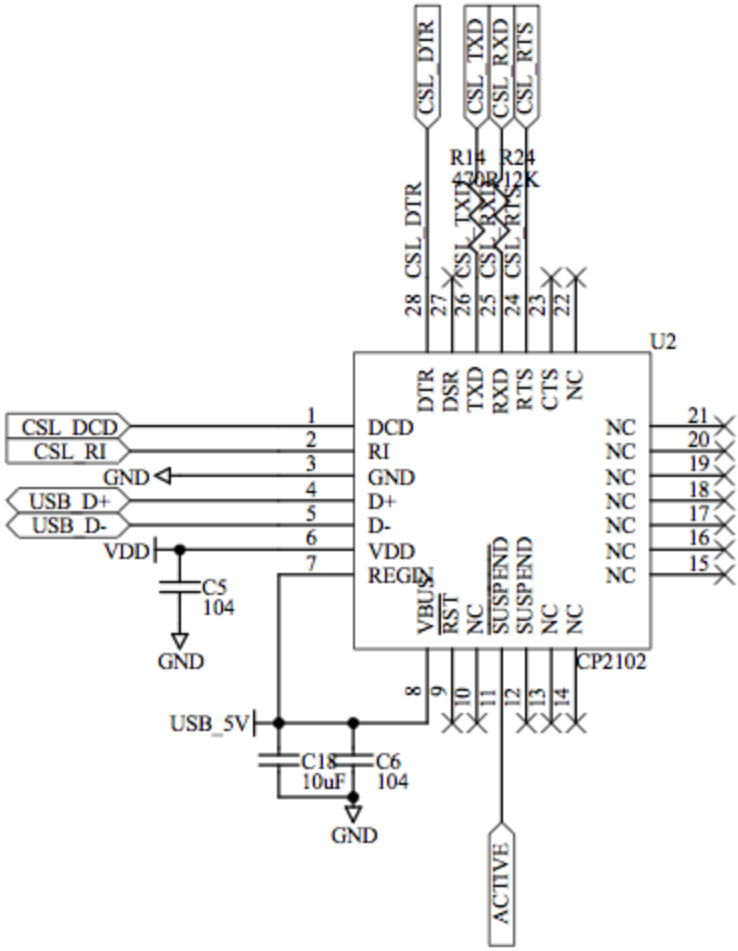
Note that this is only switched the main external 12V power line, so will provide whatever power that line provides.



CP2102 USB

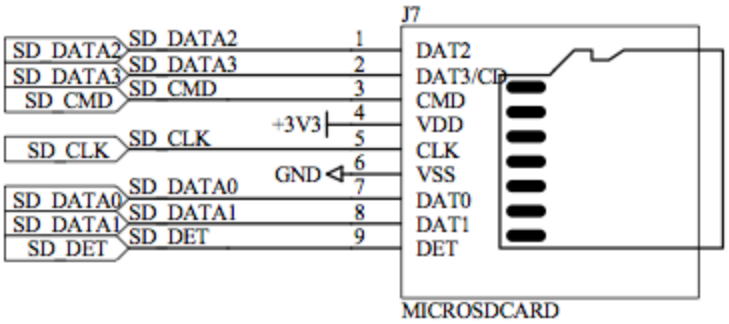
A CP2102 USB-async chip is used for connection to the ESP32 primary asynchronous console port. This is used for terminal access, low-level chip and eFuse access, as well as firmware upload.

As previously described, the CSL_DTR and CSL_RTS lines are used for control of the ESP32 during boot, and can be used to reset the system and/or enable firmware download modes.



Micro SD Card

A standard Micro SD card is available on the third SPI bus of the ESP32. This is wired in both 1 and 4 line modes, and includes a SD_DET GPIO for detection of card insertion.



ESP32 WROOM-32 Pin Assignments			
ESP32	**Function**	**OVMS**	**Comment**
1	GND1	GND	Signal Ground
2	+3V3	+3.3V	+3.3v from AOZ1280CI circuit
3	EN	ESP32_EN	ESP32 enable (reset) control
4	SENSOR_VP	ADC_IN	ADC voltage divider input
5	SENSOR_VN	SD_DET	SD CARD detect
6	IO34	SP_INT1	MCP2551 #1 interrupt line
7	IO35	SP_INT2	MCP2551 #2 interrupt line
8	IO32	EXP_1	ESP32 general purpose free expansion
9	IO33	EXP_2	ESP32 general purpose free expansion
10	IO25	CAN_TXD	ESP32 CAN TXD to SN65
11	IO26	CAN_RXD	ESP32 CAN RXD from SN65
12	IO27	SP_CS2	MCP2551 #2 CS line
13	IO14	SD_CLK	SD CARD clock
14	IO12	SD_DATA2	SD CARD DATA2
15	GND2	GND	Signal Ground
16	IO13	SD_DATA3	SD CARD DATA3
17	SD2	FL_HOLD	W32R128FV SPI Flash HOLD
18	SD3	FL_WP	W32R128FV SPI Flash WP
19	CMD	FL_CS1	WROOM-32 4MB SPI Flash CS
20	CLK	FL_SCK	W32R128FV SPI Flash SCK
21	SD0	FL_SDO	W32R128FV SPI Flash SDO
22	SD1	FL_SDI	W32R128FV SPI Flash SDI
23	IO15	SD_CMD	SD CARD CMD

24	IO2	SD_DATA0	SD CARD DATA0
25	IO0	ESP32_IO0	ESP32 IO0 Firmware ‘boot’ line
26	IO4	SD_DATA1	SD CARD DATA1
27	IO16	MDM_TXD	TX data to optional modem
28	IO17	MDM_RXD	RX data from optional modem
29	IO5	SP_CS1	MCP2551 #1 CS line
30	IO18	SP_CLK	SPI bus CLK
31	IO19	SP_MISO	SPI bus MISO
32	n/c	n/c	
33	IO21	SP_CS3	MAX7317 CS line
34	RXD0	CSL_TXD	TXD to USB async console
35	TXD0	CSL_RXD	RXD from USB async console
36	IO22	FL_CS2	W32R128FV SPI Flash CS
37	IO23	SP_MOSI	SPI bus MOSI
38	GND3	GND	Signal Ground

CAN Buses

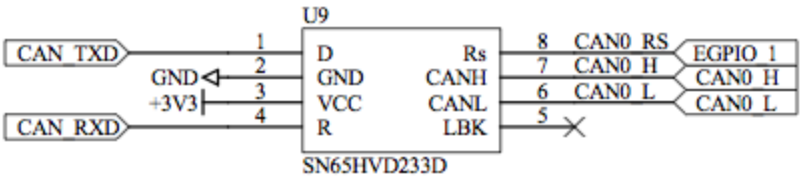
The OVMS v3 module has support for three CAN buses.

- CAN1 uses the ESP32's own on-board CAN controller
- CAN2 uses a MCP2515 SPI CAN controller
- CAN3 uses a MCP2515 SPI CAN controller

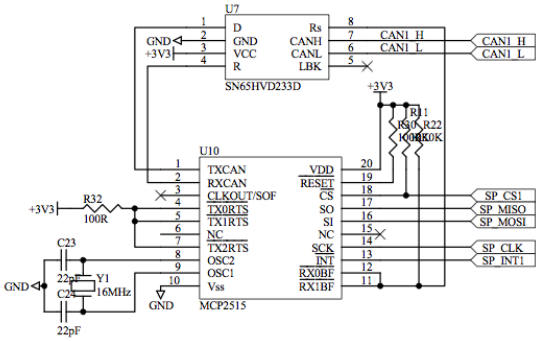
All three uses SN65 3.3v transceivers.

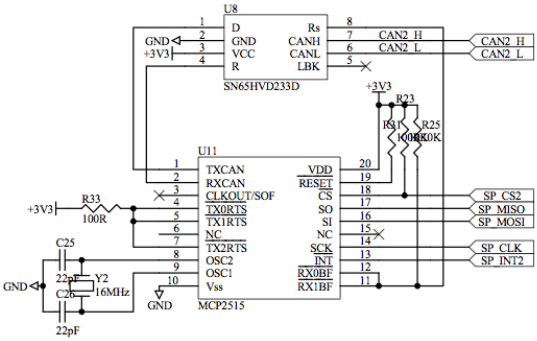
CAN1 - On Board

The CAN1 bus uses the ESP32 on-board CAN controller and an external SN65 3.3v transceiver. Power control of the transceiver is via the ESP32CAN_EN line of the MAX7317 GPIO expansion system.



CAN2 and CAN3 via MCP2515





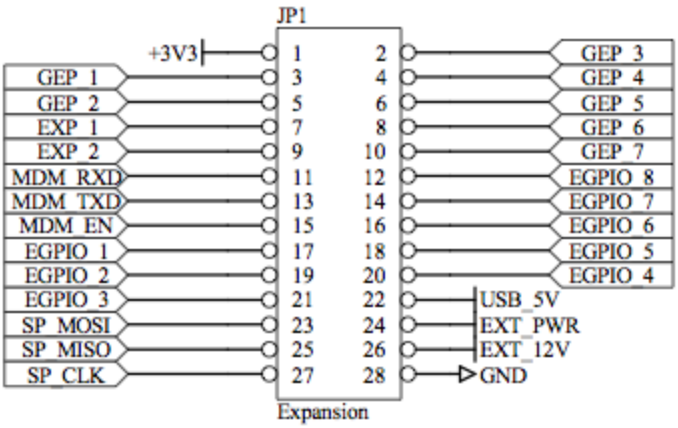
CAN2 and CAN3 are via MCP2515 SPI controllers and SN65 3.3v transceivers. Power control of the transceivers is via RX0BF/RX1BF GPIO outputs from the MCP2515 controllers.

Expansion

There are three expansion connectors on the OVMS v3 module. The connections, and wiring, for these is as follows.

Internal Expansion Bus

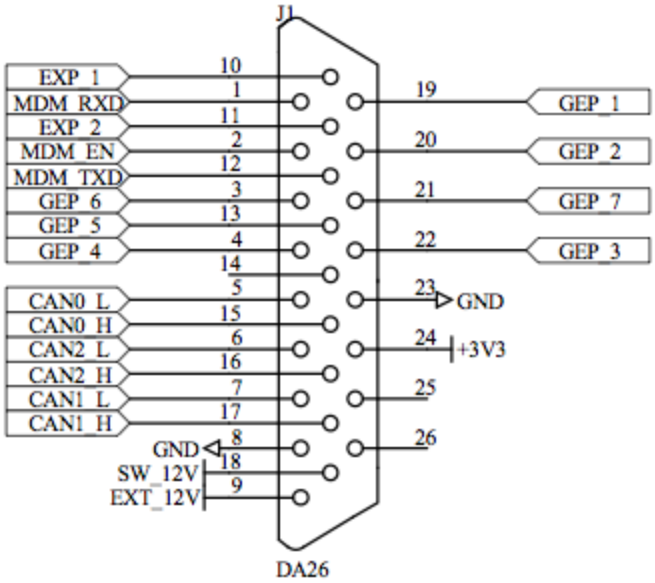
The internal expansion bus consists of two rows of 14 pins and is designed to be stackable. It exposes the EGPIO pins from the MAX7317 GPIO expansion chip, as well as available GPIO and bus pins direct from the ESP32 microcontroller on the ESP WROOM-32 module. Input/Output pins are also available from/to the DA26 external expansion connector.



DA26 External Expansion Connector

The DA26 external expansion connector is designed to be used for connection to external devices, for expansion of core functionality It exposes the GEP 1..7 pins from the internal expansion connector, as well as all CAN buses, power sources, etc.

In particular, a switched 12V power supply is available from this connector.

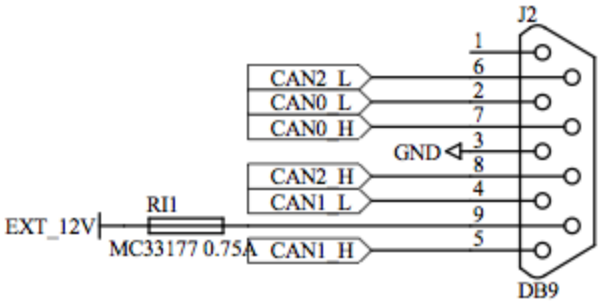


DA26 Expansion Connector Pin Assignments		
Pin	**Function**	**Comment**
1	MDM_RXD	Modem receive data line
2	MDM_EN	Modem enable control line
3	GEP_6	General Expansion Output #6
4	GEP_4	General Expansion Output #4
5	CAN0_L	First CAN bus (CAN1) low
6	CAN2_L	Third CAN bus (CAN3) low
7	CAN1_L	Second CAN bus (CAN2) low
8	GND	Ground signal line
9	EXT_12V	External 12V power supply
10	EXP_1	ESP32 general purpose free expansion
11	EXP_2	ESP32 general purpose free expansion
12	MDM_TXD	Modem transmit data line
13	GEP_5	General Expansion Output #5
14	n/c	
15	CAN0_H	First CAN bus (CAN1) high
16	CAN2_H	Third CAN bus (CAN3) high
17	CAN1_H	Second CAN bus (CAN2) high
18	SW_12V	Switched 12V
19	GEP_1	General Expansion Output #1
20	GEP_2	General Expansion Output #2
21	GEP_7	General Expansion Output #7
22	GEP_3	General Expansion Output #3
23	GND	Ground signal line

24	+3.3V	Regulated 3.3v power
25	n/c	
26	n/c	

DB9 External Expansion Connector

The OVMS v3 module includes a DB9 external expansion connector, compatible with the OVMS v2 design but exposing the extra two CAN buses available in OVMS v3.



DB9 Expansion Connector Pin Assignments		
Pin	**Function**	**Comment**
1	n/c	
2*	CAN0_L	First CAN bus (CAN1) low
3*	GND	Signal ground
4	CAN1_L	Second CAN bus (CAN2) low
5	CAN1_H	Second CAN bus (CAN2) high
6	CAN2_L	Third CAN bus (CAN3) low
7*	CAN0_H	First CAN bus (CAN1) high
8	CAN2_H	Third CAN bus (CAN3) high
9*	EXT_12V	External 12v power supply

* OVMS v2 compatible pin

Conclusions