OBD2 Handheld using a Raspberry Pi

With new diagnostic software

By **Thomas Beck** (Germany)

The Pi-OBD-HAT is an off the shelf add-on board for the Raspberry Pi which converts the tiny computer into a dedicated OBD2 diagnostic tool. One disadvantage was the need of a terminal program running on a PC for control. A much more convenient and neater solution is provided by the HHGui OBD2 diagnostic software.

The GUI software HHGui is a variant of the HHemu program I wrote for the OBD2 handheld emulator [1] which is a sort of Swiss army knife tool for developers working with the OBD2 analyzer 'NG/ DIAMEX Handheld Open' (HHOpen). This unit is a handheld OBD2 diagnostic tool published in September 2009 in *Elektor Magazine* [2]. The OBD2 analyzer NG employs the DIAMEX DXM OBD2 module [3].

The development process

The software development process is a little unconventional and shows how firmware originally written for a microcontroller can be ported to a Raspberry Pi or PC, ideally without making any changes to the firmware source code.

HHEmu was originally conceived as a pure development tool to test new ver-

sions of the AVR microcontroller firmware for the OBD2 analyzer on a PC without the need to plug it into an OBD-equipped vehicle. In addition to the firmware HHEmu contains an OBD2 emulator which supplies simulated OBD2 control unit data to the firmware via a simulated DXM OBD2 module.

The next step in the development process HHEmu was provided with a serial interface to a PC. Together with a Bluetooth expansion board published in April 2010 [4] this gave the OBD2 analyzer NG a great deal of added flexibility. The firmware running on a PC within HHEmus can control a real NG analyzer and its DXM module using a serial interface to the Bluetooth adapter with the Bluetooth extension. This makes it possible to test new firmware versions on a real vehicle without first flashing the firmware into the OBD2 analyzer. This was a real milestone in the development process! In the meantime HHEmu has become the standard diagnostic software for DXM-based diagnostic interfaces. In addition the OBD2 emulator in HHEmu can now act as a data generator to test OBD2 diagnostics tools produced by other manufacturers or for the development of new OBD2 software. As well as the improved test possibilities for firmware development there are also other benefits of porting to a PC. With each future firmware update the diagnostic software automatically receives the functionally identical update.

Now HHEmu has been ported to the Raspberry Pi. After all the changes necessary for operation with the Pi-OBD hat and the removal of any OBD2 analyzer-NG specific menus the first version of HHGui is ready for publication.

Operational modes

We can say, because of its development path that HHGui is essentially OBD2 diagnostic software with a user interface that mimics the look of the OBD2 analyzer NG as well as internally running its firmware adapted to the PI-OBD module. The current range of the firmware functions available are given in the **Table**.

HHGui presents the OBD2 user menus and data on a simulated LCD display scaled to the larger graphic display of the RPi. This is a compromise so that in addition the official 7" touchscreen recommended for the RPi it also supports displays as small as 320x240 pixels. A modification of the Pi-OBD module is necessary as described in the first project update of the Labs-project [5].

The second level of functionality implemented in HHGui is the OBD2 simulator which can be used to represent the Pi-OBD module and up to eight configurable OBD2 controllers. This feature allows you to test the full range of software functions without the need for a Pi-OBD module or vehicle. Plugging into a real vehicle will also not give you access to all the possible OBD2 functions and subfunctions because many of these are exclusively related to the type of engine fitted (i.e. petrol (gas) or diesel powered vehicles). Another bonus is that this simulator is completely harmless, there's no chance of accidentally deleting any of the vehicle's OBD2 data. The range of OBD functions provided by the simulator is identical to the range of functions in the firmware. Commands to control the OBD2 simulator entered via the keyboard are described in the Labs HHEmu project [1].

What's possible using OBD2 diagnostics?

So returning to the main functions of the OBD2 diagnostic software, what can we expect to achieve with the services available via the OBD2 interface?

When you take a close look at the ISO 15031-5 [6] specification (or the identical SAE J1979 [7]) it becomes clear that via the OBD2 port we can read emission-relevant information and trouble codes and partially delete them. This information is mainly generated by the vehicle's ECU. Vehicles with automatic transmission can also produce OBD2 trouble codes from the Power-train Control Module. Less often the vehicle will be fitted with additional control systems or even multiple ECUs in the case of hybrid vehicles. General settings such as automatic door locking or inhibiting the warning gong sound for seat belt reminder cannot be made via the OBD2 connector. With the exception of the Malfunction Indicator Lamp (MIL) none of the other fault indicator lamps on the instrument cluster or the service warning lamp or service interval can be reset. These actions can only be performed using the manufacturer's diagnostic equipment which is usually specific to the vehicle model.

That may seem a little disappointing at first. The advantage of this manufacturer standardized OBD2 diagnostics is that now after 20 years since its introduction in the US (and since the year 2000 for petrol-powered vehicles in Europe), it works with almost every vehicle. Since 2008 the proliferation of different protocols for transferring OBD2 data has been drastically reduced so that now only the CAN protocol can be used.

Take a closer look at all the available OBD2 sub-functions you will find an incredible amount of sensor information, counter values and other measurements (altogether more than 100) that's sure

The OBD2 functionality of HHGui

HHGui supports the OBD2 services contained in ISO15031-5. A more detailed description can be found on the project page [5], for more on the sub-functions refer to [6], [7] or [8]. The range of supported services will depend on the vehicle type and its OBD2 control unit. HHGui will interrogate the respective control unit and display the supported sub-functions.

- Support for up to eight OBD2 ECUs (Electronic Control Units) according to ISO 15765-4 [9]
- OBD2-Service 0x01: Read Live or Current Data; support also for Parameter Identifier (PIDs): 0x00...0x60, 0x70, 0x80, 0x8D sub-functions
- OBD2-Service 0x02: Read Freeze Frame Data, supports PIDs as for Service 0x01
- OBD2-Service 0x03: Read confirmed Diagnostic Trouble Codes (DTC)
- OBD2-Service 0x04: Clear DTCs, stored values, MIL status, OBD2 monitor and other data
- OBD2-Service 0x07: Read Pending DTCs
- OBD2-Service 0x09: Read Vehicle Information; support also for (InfoTypes): 0x00, 0x02, 0x04, 0x06, 0x08, 0x0A, 0x0B
- OBD2-Service 0x0A: Read permanent DTCs

to make you want to explore further. Apart from the sensor information you would expect to have access to from the Engine Control Unit you can also read data originating from other equipment on the vehicle such as the brake control module.

In addition to the raw OBD2 data some values undergo processing before they are displayed on the instrument panel. Some examples are:

- value smoothing (outside air temperature, fuel gauge and generally any instrument with a pointer driven by a stepper motor);
- values with a plateau function with a limited range of interest (coolant temperature);
- values with added offsets, to provide a margin for error and ensure that displayed values will not allow the driver to unknowingly exceed any legal restriction or simply to comply with the manufacturers recommended limits (speedometer, engine RPM).

Data supplied from the OBD2 interface can also be raw sensor values without any processing so we can read the actual speed rather than the value displayed on the speedometer.

There is also data that you might be interested in reading which the manufacturer does not display such as oil temperature or turbocharger boost pressure. Engine load values may also be of use.

The average user might also be interested to know the state of readiness of the vehicle for its next inspection and test (the annual MOT in the UK, or vehicle safety inspection in the US). To get this information it's necessary to run all the OBD2 test programs (OBD2 monitors) at least once after fault conditions have been erased. This condition can be checked in HHGui when in Inspection/Maintenance readiness menu the status of all the OBD2 monitors is displayed as '… monitoring ready: YES'.

In addition to this easily understandable OBD data we also have more esoteric information such as lambda probe readings more of interest to the specialist. A list of all the available Parameter IDs (PIDs) is too long to include here. The latest full-version of the official specification is available at considerable expense, as an alternative it's worthwhile taking a look at the Wikipedia entry [8].

Now we turn to a subject which is of more interest to developers.

HHGui: Structure and Function

Functional Principle

The firmware is executed in a thread — all inputs and outputs of the firmware are processed and responded to in additional threads. All of the microcontroller registers and interrupts need to be simulated.

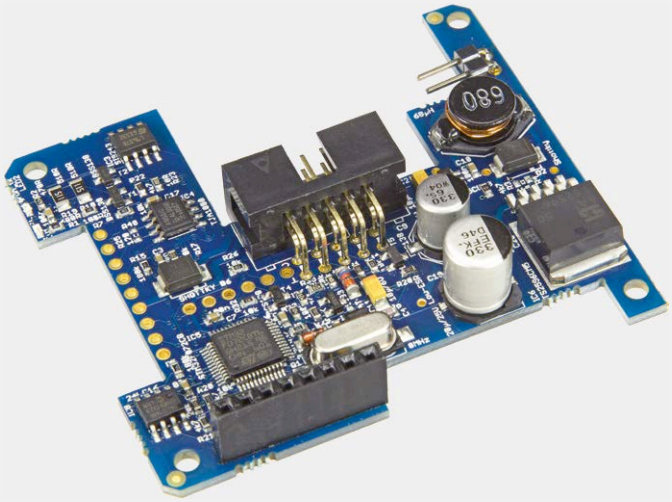
Actual implementation

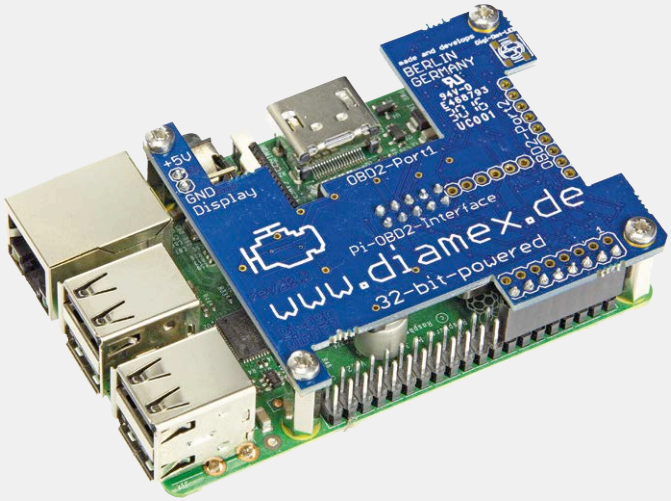
HHGui consists of six threads. **Figure 1** is a simplified block diagram showing the interaction between the threads. The source files use different names as a result of the program development history. The Pi-OBD module thread is called

PI-OBD: The Hardware

The PI-OBD-HAT [10] from Diamex turns a Raspberry Pi into an OBD2 diagnostics adapter. Fitted with the 7" touch screen you have a handy and powerful stand-alone diagnostic tool for vehicles with an OBD2 interface. The RPi and its display are powered directly by the vehicle's own 12 V supply.

Any of the Raspberry Pi variants can be used, but the RPi 2 Model B or RPi 3 Model B are preferred owing to their higher processor speed. The PI-OBD hat uses just eight pins of the RPi headers to provide power, connections for serial data interface and reset signal to activate the PI-OBD bootloader. Serial communication takes place using the TxD and RxD pins at 115,200 Baud with 8N1.





Plugging it all together

Plug the Pi-OBD hat into the Raspberry Pi headers. The 8-way header socket plugs onto eight of the I/O pins along the edge of the RPi board. The hat is held in place using the threaded spacers and screws supplied.

The Raspberry Pi is powered by the vehicle battery via the OBD2 plug. No other external power source is required unless

DxmThread and in the Firmware for den OBD2 analyzer NG HhopenThread. The shorter names are used in the following description.

The NG-Analyzer firmware is executed in the HhopenThread. This thread communicates with other threads in various ways. Display data for the LcdThread is passed via the SPDR register; RGB values for the LED back light are passed to LcdThread via registers OCR1A/B/C. The UDR0 register is used to pass AT commands, OBD requests and responses to the DxmThread. A firmware reboot is made via the WDTCR register or from MainThread of HHGui. Key presses from the MainThread are received in the PINA register. HhopenThread is controlled via the Interrupt service routines for CounterThread and DxmThread. Details are given in the descriptions of the other five threads.

The MainThread takes care of evaluating command line parameters, initializing the serial interface, generation of other threads and at the end of the program

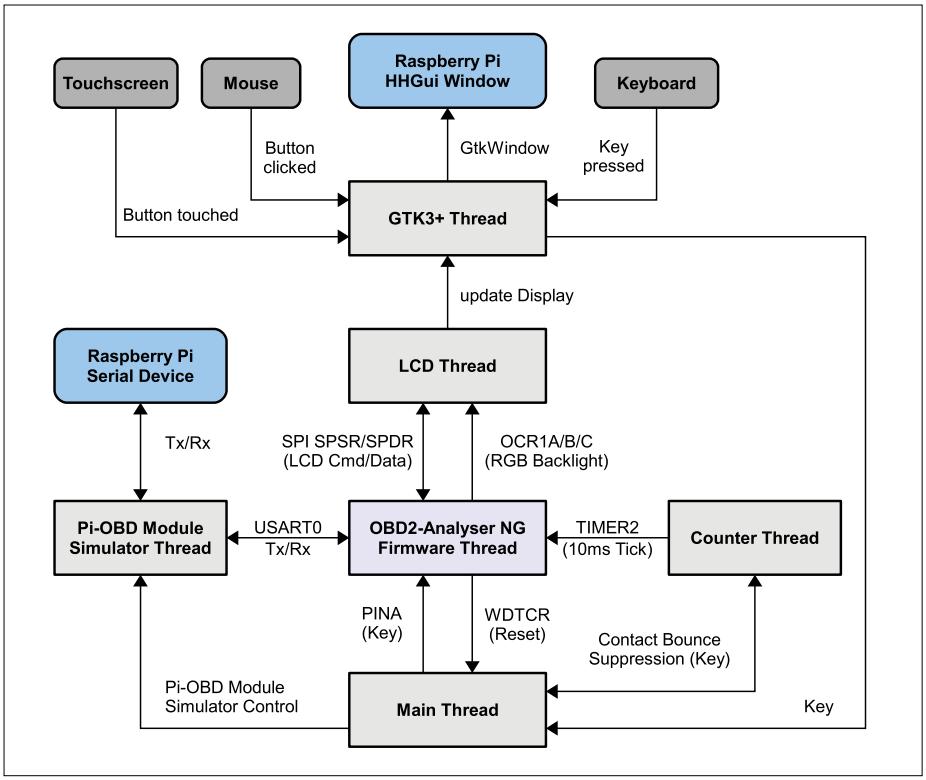
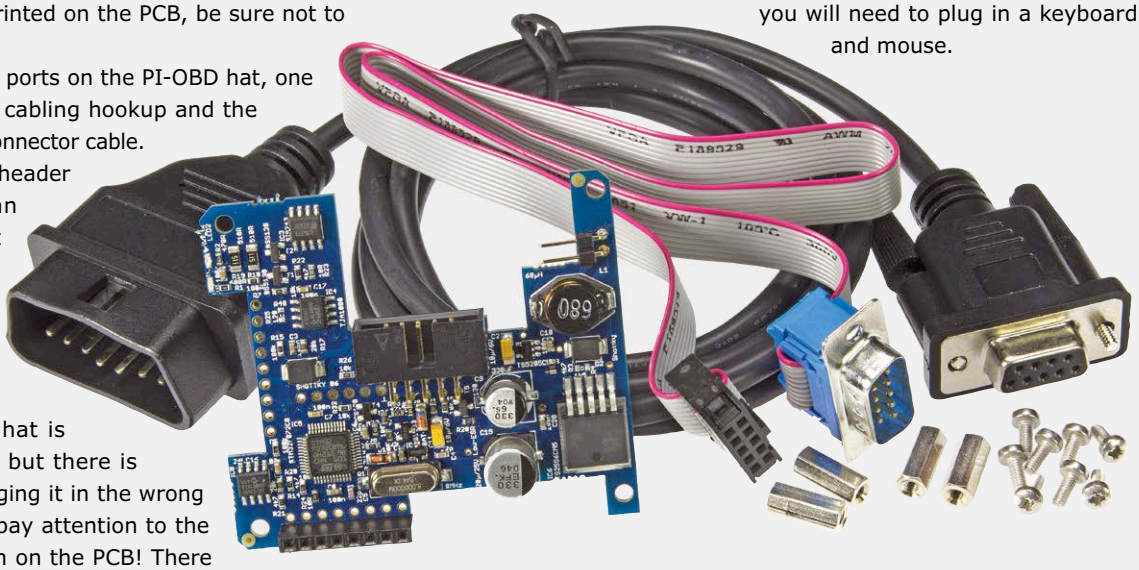


Figure 1. HHGui: Block diagram.

you are testing or configuring the unit away from the vehicle when it will then be powered via its USB connection. Power for the 7" Raspberry Pi display is provided by the two-pin header on the hat. The photo is from under the RPi with the OBD hat fitted and the display PCB above. A description of the header pin connections is printed on the PCB, be sure not to get them mixed up!

There are two OBD2 ports on the PI-OBD hat, one is suitable for 'free' cabling hookup and the other with a sub-D connector cable.

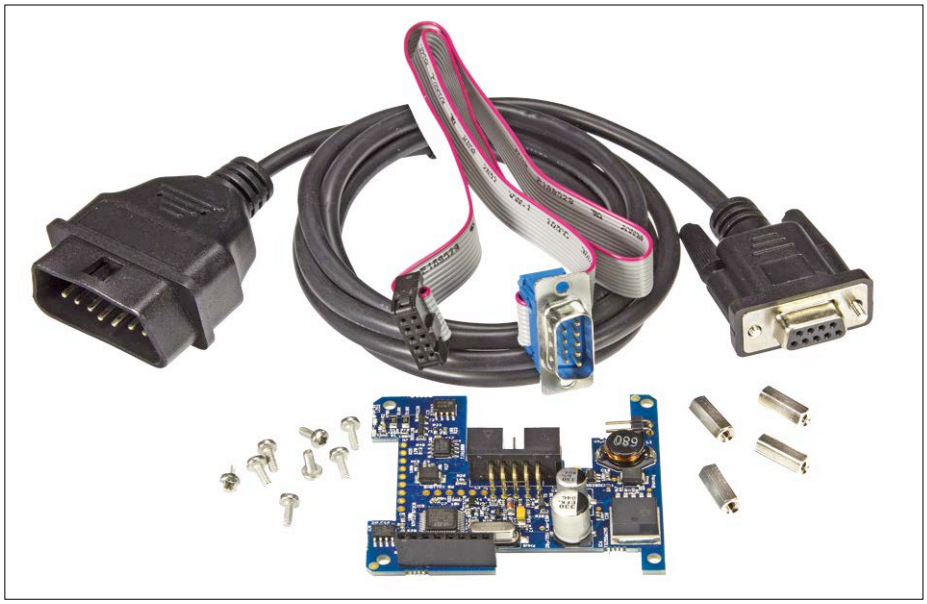
A single-row 9-way header plug (or socket) can be used to connect to Port 2 along the board edge. The port 2 connector has the advantage it can be plugged in and out whilst the hat is attached to the RPi but there is also the risk of plugging it in the wrong way round. Always pay attention to the printed identification on the PCB! There can be no confusion on port 1 because the connector on the board has a detent position and the 9-pin sub-D connector at the other end of the cable can only be connected one way round. The flat band connector on this port must be inserted before the hat is assembled onto the RPi, it can't be plugged in or withdrawn without first releasing the hat.



Fire it up

To begin using the PI-OBD HAT you will need some basic knowledge of the Raspberry Pi environment, the use of Linux and the software installation. For the purposes of configuration and operation, unless you are using a touch screen with the RPi, you will need to plug in a keyboard and mouse.

The first exercise now is to enable the RPi's serial interface but a slightly different procedure is necessary for each different RPi model and space limitations here makes it necessary to provide this information via this link [10]. The unit will be ready for use after a functional test using a terminal program.



returns these resources. The thread also evaluates key press information sent by GtkThread in an endless loop. It is necessary to differentiate between the keys used to control the OBD2 simulator and the Up/Down/ESC/OK keys used for the HhopenThread. For these keys used by the HhopenThreads it is necessary for the firmware to ignore multiple contacts produced by contact bounce. The key press must be present for 40 ms in the PINA register in HhopenThread to be recognized. This 40-ms window is measured using the CounterThread. Keys used only for the OBD2 emulator produce changes in settings in the DxmThread.

The loop that evaluates pressed keys terminates when a watchdog reset is triggered from HhopenThread (by holding down the ESC key) or by terminating HHGui using the Q key.

LcdThread evaluates the commands and display data received via the simulated SPI interface from the HhopenThread and translates them into commands for the GTK3+ graphics library. The LED backlight RGB values in the OCR1A/B/C registers define the displayed colors together with the contrast setting (via the SPI interface). The LCD in the OBD2 analyzer NG uses an ST7565R display controller so LcdThread contains an ST7565R simulator for the ST7565R commands used by the Firmware.

The GtkThread is necessary because the GTK3+ main loop (function *gtk_main()* in the GTK3+ library) blocks the rest of the gtk code. GTK3+ drawing functions should only be called from inside GtkThreads or in context of the GTK3+ main loop. When LcdThread requests a display update, it installs a callback-function, which will be called later from the GTK3+ main loop. The GtkThread evaluates all kinds of input events and generates the Up/Down/ESC/OK button presses for the HhopenThread or key input events to control the OBD2 emulator.

The CounterThread is primarily used to simulate the interrupt generated by Timer2 which occurs every 10 ms. In response the `TIMER2_COMP_vect()` Interrupt Service Routine (ISR) is called every 10 ms. Apart from that it is used in the MainThread in contact debouncing as described earlier.

In the OBD2 software mode the DxmThread acts as an interface adapter and just transfers the data unchanged. The received data in the UDR0 register (USART0-I/O data register) which may be AT commands for the Pi-OBD module or OBD2 requests, are sent to the real PI-OBD module using the configured serial interface. After each byte is read the firmware is notified that the byte has been sent via a call of the `USART_UDRE_vect()`ISR indicating that next byte can be written to the register. The reply from the Pi-OBD module is received through the serial interface and passed to the firmware via register UDR0. The reply (OK, Pi-OBD module prompt character >, error or OBD2 data) is handled bytewise in the same way by writing to the UDR0 register and a call to the `USART0_RX_vect()`ISR tells the firmware that the register has been read.

In OBD2 emulator mode the DxmThread emulates the Pi-OBD module and one or more OBD2 controllers. There is no connection to the real serial interface. ⏮

(160204)

Web Links

[1] HHEmu: www.elektormagazine.com/labs/firmware-update-and-emulator-for-obd2-analyser-ng-wireless-obd2

[2] OBD2-Analyser NG: www.elektormagazine.com/magazine/elektor-200909/19167

[3] DXM-Modul (German!): www.diamex.de/dxshop/DIAMEX-DXM-OBD2-Modul

[4] Bluetooth for OBD-2: www.elektormagazine.com/magazine/elektor-201004/19297

[5] HHGui: www.elektormagazine.com/labs/obd2-for-raspberry-pi

[6] www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=66368

[7] http://standards.sae.org/j1979_201408/

[8] https://en.wikipedia.org/wiki/OBD-II_PIDs

[9] www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=67245

[10] Pi-OBD-HAT: www.elektor.com/pi-obd-hat-obd2-module-for-raspberry-pi