



Dr. Lauer IT

ECU emulator Studio

User manual

Emulate ECUs for diagnostic development and testing

Studio, runtime and command-line tools

Version 1.5.967

<https://www.vanille.de/apps/ecumulator-studio/>

Contents

Start Here	4
Your First Emulation	5
Choose Your Starting Point	7
Read from Vehicle	8
Connect Your Tester	10
Edit, Apply and Retest	12
Protocols	14
Identifiers (DIDs)	18
DTCs (Diagnostic Trouble Codes)	20
Dynamic Values	22
Sessions and Timing	25
Security Access	27
Routines	29
Custom Services	31
Internal Bus	34
External CAN	36
HSFZ	40

DoIP	41
ELM327/STN	43
ECUconnect	46
ENET Service Broker	47
K-Line on ECUconnect	49
Writing Vehicle and ECU Specs	51
The ecum Runtime	61
The ecp Remote Control	65
Vehicle Scan Reference	69
Vehicle Files	72
ECU Spec Files	76
Demo Vehicles	80
Example Spec Files	82
Troubleshooting	91

Start Here

This manual for **ECUlator Studio 1.5.967** takes you from a first diagnostic reply to a reusable vehicle model. Begin with the included demo; you need no vehicle hardware or JSON knowledge for the first exercise.

The workflow

Choose a vehicle → **connect a tester** → **observe a reply** → **edit** → **apply and retest** → **save the model and evidence**.

Studio edits the model. The **runtime**, `ecum`, answers your tester using the applied model. A **service** connects the tester to that runtime over CAN, DoIP, HSFZ or another supported link. Studio manages a local runtime or controls one on another host; the adapters belong to that runtime's host.

A **vehicle spec** contains ECUs and their data. Each ECU has one or more **handlers**: UDS, KWP2000 or OBD-II. A service determines how a tester reaches the ECU; a handler determines its answers.

Keep three states distinct: the **editor draft**, the **saved file**, and the **applied runtime model**. Save preserves a file. **Apply Spec** changes the running model. The editor's reply preview is a calculation; a fresh request from the tester verifies the result.

Choose your path

Your goal	Start with
Get a working example	Your First Emulation
Use a vehicle, capture, database or blank model	Choose Your Starting Point
Capture a connected vehicle	Read from Vehicle
Reach the model with your tester	Connect Your Tester
Make and preserve test variants	Edit, Apply and Retest
Write JSON or automate	Writing Vehicle and ECU Specs , The ecp Remote Control

New users should complete the first exercise, then follow the workflow chapters for their own project. Consult the later configuration, service and format references when a task needs them.

Scope and notation

A scan or import supplies observed data. It cannot infer inaccessible ECUs, security algorithms or every future response. Review the result before using it as a test fixture. This is a guide to Studio, with diagnostic background assumed in the protocol references.

UI instructions use English labels; Studio also offers German. Requests are hexadecimal diagnostic bytes without transport framing unless stated. These same chapters appear in **Help** → **Documentation** and **Help** → **PDF Manual**. Next: [Your First Emulation](#).

Your First Emulation

Goal: read 90 °C from the included BMW demo, change it to 80 °C in Studio, and see the changed answer on your tester. This short exercise needs no vehicle or CAN adapter. It establishes the same cycle you will use with a scanned or hand-authored vehicle.

Before you start

You need Studio, a valid desktop emulation license, and an ELM327-compatible tester that can connect over TCP. A terminal alternative is shown below. Activate the license under **Help → License & Updates...** if necessary.

In **Settings → Sidecar Target**, select **Managed local ECUlator++** and wait for the runtime to connect. This runs the engine on the same computer as Studio. Keep **External CAN** disabled for this exercise. If a previous vehicle has unsaved work, save it before loading the demo.

1. Load the example

Open **Demo Vehicles** and choose **BMW Motorrad — 1250 Boxer** in the CAN group. Expect one logical ECU, **BMS-X**, with UDS and OBD-II handlers. The CAN group describes the demo's addresses; you can reach it through the ELM327 service without attaching a physical adapter.

The vehicle and ECUs are on the left, the selected ECU's data is in the centre, and runtime/service controls are on the right. Select BMS-X and its **OBD-II → Live data** tab. Find **Engine coolant temperature**, PID 05. Its configured constant is **90 °C**.

Checkpoint: the demo is visible in the editor. This alone does not mean a tester can reach it yet.

2. Make it available to a tester

In the service panel, enable **Internal Bus** and **ELM327**. Keep the vehicle's transport profile at **ISO-TP (Classic)**. Click **Apply Spec** if there are pending changes; if it is disabled and there is no error, there is nothing new to apply. Wait for the runtime to report the enabled service without an error.

Connect your ELM327-compatible tester to **127.0.0.1**, TCP port **35000**. The port is configured under **Settings → Services → ELM327**. Use ISO 15765-4 CAN, 11-bit, 500 kbit/s. These are the default service settings; use the configured port if it was changed. Request coolant temperature from the tester's live-data view.

Checkpoint: the tester displays **90 °C**. This is the first verified reply from the running model. If connection fails, confirm the service is enabled, the license is active and the tester uses 127.0.0.1. See [Connect Your Tester](#) for a tester on another computer.

Terminal alternative

On macOS, open a terminal and run:

```
/usr/bin/nc -c 127.0.0.1 35000
```

The built-in macOS `-c` option sends the carriage-return line endings the adapter expects. Other netcat implementations use different options; use a TCP terminal configured to end each command with a carriage return there. Wait for the `>` prompt, enter one line at a time, and wait for the next prompt before continuing:

```
ATE0
ATL0
ATS1
ATH0
ATSP6
0105
```

The setup commands answer `0K`; `0105` answers `41 05 82`. This is the 90 °C value, with headers disabled: this PID encodes temperature as the value plus 40, so $90 + 40 = 130$ (82 in hexadecimal). Keep the connection open for the next step. Use either this terminal or your tester app so their requests do not interfere with one another.

3. Change one value and retest

1. In Studio's **OBD-II** → **Live data**, double-click the PID `05` row to open its inspector.
2. Keep **Behavior** → **Constant**, select **Values**, and change **Value** from `90` to `80`. Finish the edit by pressing Enter or leaving the field.
3. Check **Reply preview**: its payload should now be `41 05 78`. This is a local calculation of the draft, not an observation from the tester.
4. Click **Apply Spec**. Wait for it to finish without an error.
5. Request coolant temperature again on the tester, or enter `0105` again in the terminal.

Checkpoint: the tester now shows **80 °C**, or the terminal answers `41 05 78`. If it still shows 90 °C, request a fresh value and check that Apply completed on the runtime to which the tester is connected.

4. Save and finish

Choose **File** → **Save As...** and save your working vehicle with a name such as `coolant-80.json`. The bundled demo remains unchanged. Close the tester connection and switch off **ELM327** when finished. Use Ctrl-C to close the terminal connection.

You now have a saved test model and have verified one deliberate change over a diagnostic connection. Next, [Choose Your Starting Point](#) for your own project, or go straight to [Read from Vehicle](#) if you have a vehicle connected. [Edit, Apply and Retest](#) explains how to keep the draft, saved file and running model in step.

Choose Your Starting Point

Goal: choose the shortest path from the material you have to a model you can test. If you have not yet seen an emulated ECU answer, complete [Your First Emulation](#) first.

What do you have?

Starting material	Choose	What you will need to review
A connected vehicle or ECU	Read from a vehicle	Which ECUs answered, which data was read and what was inaccessible
Recorded diagnostic traffic	Import from CAN logs	Address pairs, useful conversations and the requests the capture never made
An OEM diagnostic database	Import from ODX/PDX	ECU variant, actual values, addresses and behaviour not described by the database
A small, known test case	Build from scratch	The addresses, handlers and responses your tester needs
A suitable bundled example	Demo Vehicles	Which sample values and assumptions need changing for your test

The first four choices are methods in **File** → **New...**. Demos are available from their own menu. If you already have a spec, use **File** → **Open...** or drag its .json, .vehicle or .ecuvehicle file into Studio. Save your current work before replacing it; Studio asks before discarding edits or stopping a running session.

Start from a connected vehicle

Vehicle Scan turns observed diagnostic answers into an editable spec. CAN uses an adapter on the runtime host; DoIP and HSFZ use an Ethernet connection. Start with Standard inspection and keep the scan report with the spec so you can distinguish captured data from later edits. Follow [Read from Vehicle](#) for the complete procedure.

Start from recorded traffic

Choose **Import from CAN logs** and select the capture files. The importer groups diagnostic conversations by ECU and function. Review the selected conversations and enable only the blocks relevant to your test. You can omit individual request patterns; broadcast fan-out rows appear as advanced imports.

Checkpoint: the imported ECUs use the intended request/reply addresses, and a response you care about appears in the editor. A log tells you what happened during that recording; it cannot supply answers that were never requested. Compare your first emulated reply with the capture.

Start from ODX/PDX

Choose **Import from ODX/PDX**, load the database and select the ECU variant. Review the generated identifiers, routines, security levels and custom services. A database may describe the service layout without supplying usable values, security algorithms or your bench's addresses.

Checkpoint: the variant and addresses match your intended ECU, and the first response to test has concrete data. Replace starter data before relying on the model for a regression test.

Build a small model yourself

Choose **Build from scratch**. Name the vehicle, choose the starter ECU's handler and addresses, and add the response your tester needs first. Get that one request working before adding fault memory, sessions or dynamic values. Use [Demo Vehicles](#) when an existing example is close enough to adapt. For direct JSON editing, use [Writing Vehicle and ECU Specs](#).

You now have an editor draft, or a clear route to capture one. Next: [Connect Your Tester](#), then [Edit, Apply and Retest](#). Every starting method leads to the same vehicle/ECU model; you can refine it in Studio and save it as inline JSON or a vehicle with referenced ECU files.

Read from Vehicle

Goal: turn the diagnostic answers of a connected vehicle into a saved, editable model, with a report showing what was captured. Vehicle Scan in Studio 1.4 supports CAN, DoIP and BMW HSFZ.

Here, Studio acts as a diagnostic reader. Later, when you emulate the saved spec, the runtime acts as the vehicle and answers your tester. Keep these two roles separate: discovering a real gateway does not start an emulation.

1. Connect the vehicle to the runtime host

Studio needs a connected runtime that supports vehicle analysis. The connection belongs to the **runtime's computer**, which may be different from the computer displaying Studio. Attach the CAN adapter or Ethernet cable there, switch on the vehicle's ignition and open **File** → **New...** → **Read from a vehicle**.

Connection	Select in the dialog	Check before starting
CAN	CAN adapter and bitrate; CAN FD where applicable	The adapter is on the runtime host and the bitrate matches the vehicle
DoIP or HSFZ	An Ethernet interface, or Automatic	The vehicle is connected to an active interface on the runtime host

Ethernet discovery runs periodically while this dialog is open in DoIP or HSFZ mode. **Automatic** searches all active Ethernet interfaces and prefers link-local connections. You do not have to enter a gateway address. One vehicle is selected automatically; if several distinct vehicles answer, choose yours by VIN, address and interface. The same gateway answering on several interfaces can still be selected automatically.

Checkpoint: for Ethernet, your vehicle appears in the selector. This confirms that its gateway answered discovery; the ECU scan comes next. If nothing appears, check cable, ignition, selected

interface, runtime host and firewall. Known gateway addresses can be entered manually under **Enter gateway address manually**. For CAN, ECU discovery happens after you start the analysis; there is no preliminary Ethernet-style vehicle list.

2. Choose how much to read

Start with **Standard** unless you specifically need identifiers outside its ranges. Both modes use default diagnostic sessions and read requests. They exclude security access, writes, resets, routines, DTC clearing and actuator control.

Choice	What it adds to your model	Time and when to use it
Standard	Responding ECUs, identification, supported OBD-II data and fault memory, including available freeze frames	First pass. Often seconds to minutes on a responsive bench; ECU count, timeouts and gateway behaviour can make it longer
Deep	Standard plus a search through the full UDS identifier range and additional KWP identifiers	Use when you need manufacturer-specific data outside Standard. It can take hours per ECU

A completed scan means its planned requests finished; it does not guarantee that every ECU or identifier in the vehicle was accessible. For BMW and VW group vehicles, recognised by the VIN, Standard also reads the manufacturer's identification and programming identifiers outside F1xx, such as VW coding 0600 in the Audi Q7 demo; other manufacturer data needs Deep. The exact ranges and advanced addresses are in [Vehicle Scan Reference](#).

3. Run and observe

Choose **Start analysis**. Studio shows discovered ECUs, the current phase, responses and progress. Once the request plan is known, it estimates remaining time from recent request throughput. An estimate can change as slower ECUs or unanswered requests enter the scan.

On CAN, the adapter is reserved for reading. The simulator is separated from the physical vehicle, and External CAN stays disabled after the scan. On DoIP/HSFZ, analysis uses its own TCP connection through the selected interface. Configuration changes are blocked while analysis runs.

Checkpoint: ECUs and captured data appear. Unsupported requests can return negative responses; some requests may time out. These are recorded and do not necessarily mean the whole scan failed. If a known ECU is absent, check its routing/address requirements before concluding that it is not present.

Choose **Stop and keep results** to finish early. **Continue in background** closes the dialog while the runtime continues; reopen it to see progress. Cancellation or a connection failure still permits exporting data captured so far. Starting another scan replaces the runtime's previous results.

4. Save the model and its evidence

1. Use **Save report...** to retain request/response bytes, ECU addresses, confidence, failures and limitations.
2. Use **Save and open spec...** to save inline vehicle JSON and open it in the editor. The proposed name comes from the VIN where it can be decoded; you can change it in the save dialog.
3. Check the ECU count and inspect at least one identifier you know. Review handlers marked **inferred**, because some responses alone cannot distinguish UDS from KWP.

Checkpoint: you have a spec for editing and a report explaining where its data came from. Save them before another scan or a runtime exit. Opening the export does not start its simulation.

The spec retains observed payload bytes, the reported UDS DTC format and captured OBD-II freeze-frame indexes. Valid BMW F101 software-number responses are also decoded for identification; their original bytes remain intact. Silent, sleeping or protected ECUs may be missing. Security algorithms, time-varying signals and unobserved behaviour need to be modelled separately.

5. Use the captured model

Review the captured addresses before choosing the service for your tester. DoIP and HSFZ use logical addresses; they are not interchangeable with CAN request/reply IDs. If the scan used physical CAN, reconnect External CAN only when you intend to expose the simulation on that bus.

You now have a captured starting model. Next: [Connect Your Tester](#) to read it back, then [Edit, Apply and Retest](#) to make controlled variants. For automation, use [The ecp Remote Control](#).

Connect Your Tester

Goal: send a request from your tester to the intended running model and recognise its response. Start with a demo, an imported spec or the result of [Read from Vehicle](#).

1. Confirm where the runtime runs

Open **Settings** → **Sidecar Target**. With **Managed local ECUlator++**, Studio starts the runtime on this computer. With **Network endpoint**, Studio controls a runtime already running on another computer or ECUconnect.

Item	Where it belongs
Editor and file dialogs	The computer running Studio
Diagnostic services and physical adapters	The runtime host
CAN trace files	The runtime host
Desktop emulation license	The runtime host; connecting Studio does not transfer it

A tester on the runtime's computer uses 127.0.0.1. A tester elsewhere uses the runtime host's reachable address, with a service bind setting and firewall that allow that connection. The control connection used by Studio is separate from the diagnostic service used by the tester.

2. Choose the matching service

Enable **Internal Bus**, then choose the service your tester speaks. Keep unneeded physical connections disabled while preparing the model.

Your tester connects through	Enable	Read the detailed setup when needed
ELM327-compatible TCP adapter	ELM327, default port 35000	ELM327/STN

Your tester connects through	Enable	Read the detailed setup when needed
DoIP	DoIP, default port 13400	DoIP
BMW HSFZ	HSFZ, default TCP port 6801	HSFZ
A physical CAN adapter	External CAN	External CAN
ECUconnect's TCP adapter protocol	ECUconnect service	ECUconnect
ENET remote-service discovery	ENET Service Broker and its associated service	ENET Service Broker
The physical K-Line of ECUconnect	K-Line on the selected ECUconnect runtime	K-Line on ECUconnect

DoIP and HSFZ require matching logical ECU addresses. Enabling an Ethernet service does not convert arbitrary CAN IDs into the addresses a tester expects. The bundled Audi DoIP and BMW HSFZ demos are useful starting points. With a CAN FD vehicle profile, HSFZ, DoIP and ELM327 are unavailable; consult the service chapter before changing the profile.

3. Apply and request one known value

Click **Apply Spec** for pending changes and wait for it to finish. Connect the tester using the selected service, then read one identifier whose value you can see in the editor. For the first BMW example, this is coolant temperature as described in [Your First Emulation](#).

Checkpoint: the tester receives the expected value, and the runtime's traffic view records the exchange. A service being enabled proves that it is available; it does not by itself prove the tester selected the correct ECU.

If there is no response, work outwards from the request: correct runtime host, reachable service, enabled ECU, matching address, supported handler and configured identifier. Use [Troubleshooting](#) for the corresponding checks. For sequences or dynamic values, test the expected behaviour rather than requiring one constant byte string.

For physical CAN, also check the External CAN card's controller state. An adapter connection can remain open while its controller is error-passive or bus-off. Check termination, matching bitrate and an acknowledging node as described in [External CAN](#).

You now have a verified path from tester to model. Next: [Edit, Apply and Retest](#) to create and preserve test variants. Switch off the diagnostic service when that test connection is no longer needed; closing a Studio window alone may leave a remote runtime running.

Edit, Apply and Retest

Goal: make a deliberate change, prove that your tester sees it, and keep a reusable test case. Continue after [Connect Your Tester](#) or use the working BMW demo from [Your First Emulation](#).

1. Know which state you are changing

Action	What it changes	What to verify
Edit an ECU, identifier or fault	The editor draft	The field and reply preview show your intended data
Save / Save As	Vehicle and ECU files	The chosen file contains the version you want to keep
Apply Spec	The running runtime's model	A new tester request returns the intended response
Toggle a service or enable/disable a live ECU	Runtime availability	The tester can or cannot reach the intended ECU/service

Autosave helps restore the workspace after restarting Studio. Use an explicit Save for the portable file you want to share or keep as a fixture. Live ECU enable/disable does not rewrite the saved spec; loading or applying a model resets that runtime state.

Auto-apply after loading specs is an optional setting. Loading a file may therefore affect the runtime when services are enabled. Check your active runtime before opening another model. The scan's **Save and open spec...** path opens the captured result without starting its simulation.

2. Make the smallest useful change

For the BMW exercise, keep a copy of the 90 °C baseline and a second file for the 80 °C variant. Change one value, then compare a fresh tester response. This makes a failed test easier to explain than changing many fields at once.

The test needs	Edit	Reference
A different VIN, software number or sensor value	The corresponding handler's identifier/data entry	Identifiers
A fault or freeze-frame scenario	The handler's DTC or freeze-frame data	DTCs
A changing signal or sequence of replies	The value's generator or sequence	Dynamic Values
Session-dependent access or timing	Session rules and timing	Sessions and Timing
Unlocking or a routine	Explicit security/routine definitions	Security Access, Routines
A response outside the built-in handlers	A custom service	Custom Services

Use the handler your tester actually reads. A vehicle can store a VIN both at vehicle level and inside several handlers; changing one does not guarantee that every other copy changes. Preserve intentional differences in a captured vehicle and verify the specific request your tester uses.

3. Apply, then compare at the tester

After completing the edit, click **Apply Spec** and wait for success. A reply preview is computed from the draft; only a new diagnostic request confirms the applied model. Request the value again or refresh the tester's display.

Applying resets diagnostic sessions and restarts sequences at their first step. Time-based generators continue on the runtime's clock. Account for these behaviours when comparing repeated runs. A second client can also advance a sequence, so use one tester for a controlled comparison.

Checkpoint: the change appears in the received reply, with the expected address and payload. If it does not, confirm the target runtime, Apply result, handler and tester cache before changing more configuration.

4. Save the test case and evidence

Save the verified model under a descriptive name. Keep a baseline alongside variants and note the request and expected reply in the spec's comment/info fields. For scans, preserve the original report separately; later edits do not change that record of the vehicle's answers.

Opening a different vehicle can replace an active session. Studio prompts when work is unsaved or a session is running; save the files you need before accepting. If your vehicle references separate ECU files, keep them together with their relative paths. [Vehicle Files](#) describes both referenced and inline storage.

Record traffic when a result needs explaining

Enable **Settings** → **Sidcar** → **Managed Local CAN Trace** to append every frame the runtime sees — in both directions, including traffic no emulated ECU handles — to a candump log. The runtime names each file after the session and the time, so the setting takes a directory; the default is `~/Documents/CANsole/Traces`. The **Trace** card in the status bar shows whether recording is active; click it to open the recording actions.

Loading another vehicle starts a new trace file named after it, so a session that works through several vehicles leaves one readable file per vehicle. Click the **Trace** card to write a marker into the current file — the dialog is prefilled with the time and you can replace it with whatever names the run. The marker is a comment line in order with the frames around it, which makes it a reliable place to split the file later. The same dialog can reveal the folder.

This is only available for a **managed local** runtime, because the option is passed when Studio starts the process and the directory is resolved on the machine running it. A network endpoint has to be started with `--can-log` on its own host. Changing either setting restarts the managed runtime, since the option is read once at startup.

Open the resulting file in CANcorder to compare requests and replies. If the runtime reports dropped frames, the trace is incomplete; account for that when interpreting missing traffic.

5. Finish or repeat

Disconnect the tester and disable services no longer needed. A network runtime can keep running after Studio disconnects; a managed local runtime follows the **Managed Local Detach** setting. Do not rely on closing the editor to end every diagnostic service.

You now have a saved model and evidence for its intended response. Repeat the cycle for another test, consult the configuration references for the required behaviour, or use [The ecp Remote Control](#) to automate an established workflow. [Writing Vehicle and ECU Specs](#) takes the same BMW example deeper into the JSON model.

Protocols

ECUlator supports three diagnostic protocols: UDS, KWP2000, and OBD-II. Each protocol has its own service structure and configuration options.

Transport is configured separately at the vehicle level:

- `isotp` for classic CAN + ISO-TP
- `isotp_fd` for CAN FD + ISO-TP/FD
- `sae_tp20` for classic CAN + SAE TP2.0

`sae_tp20` is a bus transport profile, not a fourth ECU protocol. In practice it most often carries KWP.

UDS (Unified Diagnostic Services)

UDS is defined in ISO 14229 and is the modern standard for automotive diagnostics.

Supported Services

SID	Service	Description
0x10	DiagnosticSessionControl	Switch between sessions
0x11	ECUReset	Reset the ECU
0x14	ClearDiagnosticInformation	Clear DTCs
0x19	ReadDTCInformation	Read DTCs
0x22	ReadDataByIdentifier	Read DIDs
0x27	SecurityAccess	Unlock protected functions
0x2E	WriteDataByIdentifier	Write DIDs
0x31	RoutineControl	Execute routines
0x34	RequestDownload	Start download session
0x36	TransferData	Transfer data blocks
0x37	RequestTransferExit	End transfer session
0x3E	TesterPresent	Keep session alive

Session Types

- **Default (0x01)** — Basic read-only access
- **Programming (0x02)** — Flash programming
- **Extended (0x03)** — Extended diagnostics

UDS Custom Services

UDS configs can also define customServices for requests outside the built-in SID handlers.

Custom services support:

- strict/prefix/suffix matching
- fixed response payloads
- replay sequences with looping
- optional service actions

KWP2000 (Keyword Protocol)

KWP2000 is defined in ISO 14230 and is the predecessor to UDS, still used in many vehicles.

Supported Services

SID	Service	Description
0x10	StartDiagnosticSession	Switch session
0x11	ECUReset	Reset ECU
0x14	ClearDiagnosticInformation	Clear DTCs
0x17	ReadStatusOfDTC	Read DTC status
0x18	ReadDTCByStatus	Read DTCs by status
0x21	ReadDataByLocalIdentifier	Read local ID
0x22	ReadDataByCommonIdentifier	Read common ID
0x27	SecurityAccess	Security unlock
0x2E	WriteDataByIdentifier	Write data
0x31	StartRoutineByLocalIdentifier	Start routine
0x3E	TesterPresent	Keep alive

Identifier Types

KWP2000 uses three identifier categories:

- **Common Identifiers** — 2-byte IDs (like UDS DIDs)
- **ECU Identifiers** — 1-byte local IDs
- **Local Identifiers** — 1-byte local IDs

KWP Custom Services

KWP configs support the same customServices model as UDS. This is commonly used for manufacturer bootloader commands that do not map cleanly to standard KWP services.

For KWP-over-TP2.0, prefer standard KWP fields first (ecuIdentifiers, localIdentifiers, securityLevels, routines, sessions) and reserve customServices for behavior not covered by the standard handlers.

OBD-II

OBD-II is the standardized on-board diagnostics protocol required in all vehicles since 1996 (USA) / 2001 (EU).

Supported Modes

Mode	Description
01	Request current powertrain data
02	Request freeze frame data
03	Request stored DTCs
04	Clear DTCs and stored values
05	Request oxygen sensor monitoring
06	Request on-board test results
07	Request pending DTCs
08	Control on-board components
09	Request vehicle information
0A	Request permanent DTCs

Arbitration IDs

OBD-II uses standardized CAN IDs:

Mode	Request ID	Response ID
Standard	0x7DF (broadcast)	0x7E8–0x7EF
Extended	0x18DB33F1	0x18DAF1xx

Firewall Role and Global Actions (Studio UI)

In ECU Detail, **Global Actions** is shown only when:

- ECU protocol is **UDS** or **KWP**
- ECU role resolves to **FIREWALL**

Role matching is case/separator-insensitive, so values like FIREWALL, firewall, Fire-Wall, fire wall, and fire_wall all enable the section.

The section controls startup firewall behavior via ECU startupActions:

- lockAllOtherECUs (optional strategy, e.g. bmwNrcForbidden)
- unlockAllOtherECUs

Combining Protocols on One ECU

Many real ECUs answer more than one protocol on the same CAN node — most commonly UDS *and* OBD-II side by side, often on the very same request/reply arbitration IDs. An ECU spec file still picks exactly one kind, but the vehicle level supports this directly: add **two** `ecus[]` **entries with the same name** (one kind: "uds", one kind: "obd2"), each pointing at its own spec file (or inline block). At runtime they are merged into a single ECU that dispatches by service ID, so both protocols work — including the shared-arbitration-ID case.

```
{
  "name": "My Vehicle",
  "vin": "...",
  "ecus": [
    { "name": "Engine Control Module", "path": "engine-uds.json" },
    { "name": "Engine Control Module", "path": "engine-obd2.json" }
  ]
}
```

In Studio, create one ECU and use **+ OBD-II** in its header to add an OBD-II interpreter alongside UDS. The header shows the shared ECU name; the rows show interpreter names. Name and role are edited once for the ECU. Only missing protocols are offered. An interpreter's type is fixed: remove it and add the desired type to change the configuration deliberately.

Use the trash button beside the protocol badge in the **ECU Detail** header to remove just that interpreter. **Remove ECU** in the sidebar removes the entire ECU and all its interpreters. Existing sessions migrate automatically. Imported duplicate interpreters remain visible; remove the extra entry before saving or applying.

Give both entries matching `arbitrationInfo`/arbitration request/reply IDs to model a single physical diagnostic connection, or different IDs if the ECU genuinely separates the two protocols onto distinct addresses.

See it running: **Demo Vehicles** → **BMW Motorrad — 1250 Boxer** in the **CAN** group loads exactly this pattern (`specs/motorrad/bmw-motorrad-vehicle.json`).

Protocol Selection

Choose the protocol (or protocols — see above) based on your target vehicle:

- **Modern vehicles (2008+)** — UDS, frequently alongside OBD-II
- **Older European vehicles** — KWP2000
- **Generic diagnostics only** — OBD-II

For vendor-specific request flows in UDS/KWP, use [Custom Services](#).

Identifiers (DIDs)

Data Identifiers (DIDs) define the data that can be read from or written to an ECU using ReadDataByIdentifier (0x22) and WriteDataByIdentifier (0x2E) services.

Overview

Each identifier consists of:

- **ID** — 2-byte identifier (e.g., 0xF190 for VIN)
- **Name** — Human-readable description
- **Type** — Response format (hex, ascii, dynamic)
- **Content** — The data to return

Standard DIDs

Common UDS identifiers defined in ISO 14229:

DID	Name	Description
0xF180	Boot Software ID	Boot software version
0xF186	Active Session	Current diagnostic session
0xF187	Spare Part Number	Vehicle spare part number
0xF188	Software Version	Application software version
0xF189	Hardware Version	Hardware version
0xF18A	Supplier ID	ECU supplier identifier
0xF18B	Manufacturing Date	ECU manufacturing date
0xF18C	ECU Serial Number	Unique serial number
0xF190	VIN	Vehicle Identification Number
0xF197	System Name	System/ECU name

Content Types

Hex (Static)

Returns fixed hexadecimal bytes:

```
{
  "id": "0xF188",
  "name": "Software Version",
  "type": "hex",
  "bytes": "01 02 03"
}
```

ASCII (Static)

Returns ASCII string as bytes:

```
{
  "id": "0xF190",
  "name": "VIN",
  "type": "ascii",
  "content": "WBADE6324VBW12345"
}
```

Dynamic

Returns values that change over time, useful for simulating live sensor data:

```
{
  "id": "0x1234",
  "name": "Engine RPM",
  "type": "dynamic",
  "dynamic": {
    "min": 800,
    "max": 6500,
    "period": 5,
    "periodUnit": "s",
    "curve": [
      { "t": 0, "v": 800 },
      { "t": 0.5, "v": 3000 },
      { "t": 1, "v": 800 }
    ]
  }
}
```

See [Dynamic Values](#) for detailed curve configuration.

Adding Identifiers

1. Open the ECU's UDS or KWP workspace and the **Identifiers** tab
2. Click **Add...**, pick a standard identifier from the catalog or enter your own ID (hexadecimal)
3. Choose the representation: text, bytes, a BCD date or a sequence
4. Configure the content in the editor beside the table

Session Requirements

By default, identifiers are accessible in all sessions. To restrict access, configure session requirements in the spec file.

Read vs Write

- **Read** — Always supported for configured identifiers
- **Write** — Requires explicit configuration and appropriate security level

DTCs (Diagnostic Trouble Codes)

Diagnostic Trouble Codes are standardized fault codes stored by ECUs when they detect a problem.

Overview

DTCs are returned in response to:

- **UDS 0x19** — ReadDTCInformation
- **KWP 0x17/0x18** — ReadDTCByStatus
- **OBD-II Mode 03/07/0A** — Read stored/pending/permanent DTCs

DTC Format

UDS/KWP DTCs

3-byte format:

[High Byte] [Low Byte] [Status Byte]

Example: P0123 → 0x01 0x23 0x29

OBD-II DTCs

Standard SAE J2012 format:

[Type][System][Fault]

Prefix	System
P0xxx	Powertrain (generic)
P1xxx	Powertrain (manufacturer)
P2xxx	Powertrain (generic)
P3xxx	Powertrain (reserved)
C0xxx	Chassis (generic)
B0xxx	Body (generic)
U0xxx	Network (generic)

DTC Status

The status byte indicates the DTC's state:

Bit	Meaning
0	Test failed
1	Test failed this operation cycle

Bit	Meaning
2	Pending DTC
3	Confirmed DTC
4	Test not completed since last clear
5	Test failed since last clear
6	Test not completed this operation cycle
7	Warning indicator requested

Common status values:

- **0x09** — Stored, active
- **0x29** — Stored, not active (historical)
- **0x08** — Confirmed

Adding DTCs

UDS/KWP

1. Open the ECU's UDS or KWP workspace and the **DTCs** tab
2. In the quick entry, type the code as three hex bytes (e.g. 01 23 45); UDS and KWP DTCs have no P-code form
3. Choose a status template and click **Add** (or press Enter); the new row is selected
4. Adjust the status flags in the editor beside the table

A code that is already listed is not added twice; the existing row is selected instead.

OBD-II

1. Open the OBD-II workspace and the **DTCs** tab
2. Type the code in standard format (e.g. P0300); the list offers common codes
3. Choose the status template and the list (stored for 03/07, or permanent for 0A), then **Add**
4. Adjust the status flags in the editor beside the table

What the ECU answers

- **UDS 19** — sub-functions 01 (number of DTCs by status mask), 02 (DTCs by status mask) and 03 (snapshot identification). Snapshot and extended data records (19 04, 19 06) are not modelled.
- **KWP 18** — DTCs by status; group FFFF means all DTCs (SSF 14230-3 8.2).
- **OBD-II 03, 07, 0A** — stored, pending and permanent DTCs.

Clearing DTCs

DTCs are cleared when:

- ClearDiagnosticInformation (0x14) is received
- OBD-II Mode 04 is executed

Protocol Differences

Feature	UDS	KWP	OBD-II
DTC	3 bytes	2 bytes + statusOfDTC	2 bytes
Status	8 flags, filtered by mask	statusOfDTC	list (stored, pending, permanent)
Snapshot / extended data	Not modelled	Not modelled	Freeze frames (Mode 02)
Clear service	0x14	0x14	Mode 04

K-Line demo and freeze frames

The built-in **K-Line → AL319 OBD-II · 2 ECUs** demo has five stored and two pending codes across Engine and Transmission. Stored/pending display has been checked on the AL319. Permanent codes are configured too; the AL319 manual limits that scanner function to CAN.

OBD-II codes occupy two bytes: P0171 is 01 71, P0420 is 04 20. K-Line replies use three code pairs per message, padded with zero; the runtime chooses the transport layout automatically.

In the OBD-II **Freeze frames** tab, PID 02 identifies the DTC associated with a snapshot. Add only the measurements needed for your test. The demo supplies P0171 with 85 °C / 2250 rpm / 60 km/h and P0420 with 95 °C / 3000 rpm / 100 km/h. These fixed snapshots use request indices 00 and 01. They are not captured automatically when a fault becomes active.

Dynamic Values

Dynamic values simulate changing sensor data over time, making ECUlator useful for testing live data displays and data logging applications.

Overview

Instead of returning static bytes, a dynamic identifier returns values that:

- **Change over time** following a defined curve
- **Loop continuously** or run once
- **Scale to byte representation** automatically

Curve Editor

Click **Edit Curve** on any dynamic identifier to open the visual curve editor:

- **Drag nodes** to adjust timing and values
- **Click the curve** to add new nodes
- **Delete nodes** using the trash icon
- **Preview playback** to see the animation

Configuration

Dynamic values are defined with these properties:

Property	Description
min	Minimum value in the range
max	Maximum value in the range
period	Duration of one cycle
periodUnit	Time unit (ms, s, min)
loop	Whether to repeat (default: true)
curve	Array of time-value points

Curve Points

Each point in the curve has:

- **t** — Normalized time (0.0 to 1.0)
- **v** — Normalized value (0.0 to 1.0)

The actual value is calculated as:

$$\text{actual} = \text{min} + (v * (\text{max} - \text{min}))$$

Example: Engine RPM

```
{
  "id": "0x0C",
  "name": "Engine RPM",
  "type": "dynamic",
  "dynamic": {
    "min": 800,
    "max": 6500,
    "period": 10,
    "periodUnit": "s",
    "curve": [
      { "t": 0, "v": 0 },
      { "t": 0.3, "v": 0.5 },
      { "t": 0.5, "v": 0.8 },
      { "t": 0.7, "v": 0.5 },
      { "t": 1, "v": 0 }
    ]
  }
}
```

This creates an RPM signal that:

1. Starts at 800 RPM (idle)
2. Rises to ~3650 RPM at 30% through the cycle
3. Peaks at ~5360 RPM at 50%
4. Falls back to idle by the end
5. Loops every 10 seconds

Presets

The curve editor includes common presets:

Preset	Description
Linear	Straight line from min to max
Sine	Smooth wave pattern
Triangle	Linear up and down
Sawtooth	Linear up, instant reset
Square	Instant transitions
Idle	Constant low value
Spike	Quick peak and return

Byte Encoding

Dynamic values are encoded based on the OBD-II PID formula or custom scaling:

- **RPM** — Value / 4, 2 bytes
- **Speed** — Direct value, 1 byte
- **Temperature** — Value + 40, 1 byte
- **Percentage** — Value * 2.55, 1 byte

Use Cases

- **Live data testing** — Verify scan tool displays update correctly
- **Data logger validation** — Confirm logging captures changing values
- **Gauge testing** — Test instrument cluster visualizations
- **Simulation scenarios** — Create realistic driving profiles

Sessions and Timing

Diagnostic sessions decide which services an ECU offers. A tester switches with **10 xx** (UDS DiagnosticSessionControl, KWP StartDiagnosticSession); the ECU starts in the default session.

The Session & timing tab

- **ECU timing** at the top applies to every session that does not set its own.
- **Sessions** lists the sessions the ECU knows. Without a list, the ECU enters every session it is asked for, as older specs expect.
- **Reject other sessions** answers unlisted sessions with **7F 10 12**. The default session is always entered.

For each listed session:

Field	Effect
Timing	UDS: P2 and P2* in the 50 reply. KWP: what 83 answers while the session is active. Empty fields inherit the ECU's.
Entering requires security level	Without one of these levels unlocked, 10 xx gets 7F 10 33 .
Reachable from	From any other session, 10 xx gets 7F 10 22 .
Reply data after 50, response delay	KWP only: extra bytes and a delay for the 50 reply.

Each security level has **Available in sessions** in the Security tab: outside those sessions, 27 gets **7F 27 7F** (UDS) or **7F 27 80** (KWP). A level without sessions keeps the protocol's rule: UDS anywhere but the default session, KWP anywhere. Without a session list in the spec, the choice offers the standard sessions (UDS 01–04, KWP 81, 82, 85–87); UDS 01 is possible but outside ISO 14229-1, and Studio warns about it.

Standard sessions

Protocol	ID	Session	Source
UDS	01	Default	ISO 14229-1
UDS	02	Programming	ISO 14229-1
UDS	03	Extended Diagnostic	ISO 14229-1
UDS	04	Safety System Diagnostic	ISO 14229-1
UDS	40–5F / 60–7E	Vehicle manufacturer / system supplier	ISO 14229-1
KWP	81	Standard	SSF 14230-3
KWP	82	Periodic Transmissions	SSF 14230-3
KWP	85	Programming	SSF 14230-3

Protocol	ID	Session	Source
KWP	86	Development	SSF 14230-3
KWP	87	Adjustment	SSF 14230-3
KWP	89-F9 / FA-FE	Vehicle manufacturer / system supplier	SSF 14230-3

ISO 14230-3 itself leaves KWP session numbers to the manufacturer.

Security and session changes

Switching to another session locks security again (ISO 14229-1; SSF 14230-3 for the return to 81). The exception is a session whose required level is the one unlocked: a login that opened a session stays valid in it.

KWP ECUs differ in the order. ISO 14230-3 logs in first and then enters the programming session, which requires the login. SSF 14230-3 enters the session first and then logs in. Model the first with **Entering requires security level**; the second needs services per session, which the runtime does not model yet.

stopDiagnosticSession (20) does what 10 81 does in KWP. SSF 14230-3 dropped it; clear **Supports stopDiagnosticSession** for such an ECU, and 20 gets **7F 20 11**.

Timing parameters

Parameter	Default	Meaning
P2	50 ms	Longest time to the first response
P2*	2000 ms	Longest time after a response pending (7F xx 78)
S3	5 s	Session keep-alive; stored in the spec, not yet enforced by the runtime
Response pending every	P2* – 100 ms	How often a slow answer repeats 7F xx 78 until it is ready; kept below P2*

A slow answer (any configured delay above P2) sends 7F xx 78 at P2, repeats it at the interval, and then answers after its delay. Other requests to the ECU meanwhile get 7F xx 21 (busy, repeat request). KWP sends its first 78 after **First response pending after** (50 ms by default).

OBD-II

OBD-II has no sessions; all services are always available.

Security Access

Security Access (Service 0x27) protects sensitive ECU functions by requiring authentication before access is granted.

Overview

The security handshake follows a challenge-response pattern:

1. Tester requests seed for a security level
2. ECU returns a random seed
3. Tester calculates key from seed using the security algorithm
4. Tester sends key back to ECU
5. ECU validates key and unlocks the security level

Security Levels

Security levels are defined as pairs of sub-functions:

Request Seed	Send Key	Typical Use
0x01	0x02	Level 1 (read protected data)
0x03	0x04	Level 2 (write data)
0x05	0x06	Level 3 (routine control)
0x11	0x12	Level 9 (programming)

Odd numbers request seeds, even numbers send keys.

Seed-Key Pairs

Configure seed-key pairs for each security level:

```
{
  "level": "0x01",
  "seedKeyPairs": [
    { "seed": "AA BB CC DD", "key": "11 22 33 44", "keyResponse": "34" },
    { "seed": "12 34 56 78", "key": "87 65 43 21" }
  ]
}
```

When the tester requests a seed, ECUmulator randomly selects one of the configured pairs and returns that seed. The tester must respond with the matching key. keyResponse is optional response data appended after the positive 67 <sendKeyLevel> response.

Adding Security Levels

1. Open the ECU's UDS or KWP workspace and the **Security** tab
2. Click **Add**; the new row is selected and its editor opens beside the table
3. Enter the level (e.g. 0x01, the odd request-seed value)

4. Add one or more seed-key pairs
5. Optionally set the unlock delay, the key timeout and the sessions the level is available in

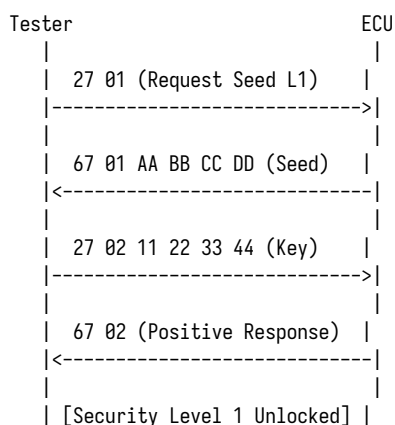
Timing

```
{
  "level": "0x11",
  "delay": 400,
  "keyTimeoutMs": 5000,
  "seedKeyPairs": [...]
}
```

- **Unlock delay** (delay, milliseconds; Studio edits it in seconds): how long the ECU takes to check a key. It answers 7F 27 78 (response pending) first, then 67.
- **Key timeout** (keyTimeoutMs; seconds in Studio): how long a seed stays valid. A later key gets 7F 27 24 (UDS) or 7F 27 22 (KWP), and the tester must request a new seed. 0 or missing means no limit.

A lockout after wrong keys (SSF 14230-3 suggests 10 seconds after two failures) is not modelled yet.

Protocol Flow



Negative Responses

NRC	Meaning
0x12	Sub-function not supported
0x22	Conditions not correct
0x35	Invalid key
0x36	Exceeded number of attempts
0x37	Required time delay not expired

Tips

- Use unique seed-key pairs to avoid guessing
- Configure realistic delays for proper tool testing
- Security unlock typically resets on session change

Routines

Routines allow diagnostic tools to execute specific functions on the ECU, such as actuator tests, calibrations, or self-tests.

Overview

Routines are controlled by:

Protocol	Request	Routine ID
UDS	31 01 (start), 31 02 (stop), 31 03 (request results)	2 bytes
KWP	31 (start), 32 (stop), 33 (request results) by local identifier	1 byte

Each routine has a unique identifier and can accept an option record after the ID.

Adding Routines

1. Open the ECU's UDS or KWP workspace and the **Routines** tab
2. Click **Add**; the new row is selected and its editor opens beside the table
3. Set the routine ID (UDS 2 bytes, e.g. 0xFF00; KWP 1 byte) and a name
4. For each response: the option record it answers, the start, stop and results replies, and an optional delay
5. Optionally restrict the routine to security levels

The table shows how many responses each routine has; the tab shows how many routines the ECU has.

Response Configuration

A routine lists one or more responses. The ECU picks the first response whose option record equals the request's; each response carries the reply data for start, stop and results:

```
{
  "id": "0xFF00",
  "name": "Erase Memory",
  "responses": [
    {
      "optionRecord": "01",
      "delay": 400,
      "startResponse": "00",
      "stopResponse": "00",
      "resultsResponse": "00"
    }
  ]
}
```

- The reply is the positive response plus these bytes, e.g. 71 01 FF 00 00 for a UDS start.
- Without stopResponse or resultsResponse, a stop or results request gets 7F 31 12 (sub-function not supported).

- delay is in milliseconds (Studio edits it in seconds). A delay above P2 sends 7F 31 78 first, see [Sessions and Timing](#).
- The option record may be written as a byte list or a hex string.

Option Record

The option record is the data after the routine ID in the request.

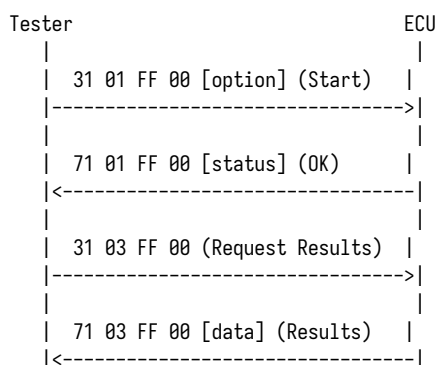
- By default the request's option record must equal a response's optionRecord; otherwise the ECU answers 7F 31 31 (request out of range).
- Set ignoreOptionRecord: true to accept any option record; the first response answers.

Standard Routine IDs

ISO 14229-1 reserves a few UDS routine IDs; everything else is up to the manufacturer.

ID	Name (ISO 14229-1)
0xFF00	eraseMemory
0xFF01	checkProgrammingDependencies

Protocol Flow



Session Requirements

Routines typically require:

- Extended diagnostic session (0x03)
- Security access at appropriate level

Negative Responses

NRC	Meaning
0x12	Sub-function not supported
0x22	Conditions not correct
0x24	Request sequence error
0x31	Request out of range

NRC	Meaning
0x33	Security access denied
0x72	General programming failure

Custom Services

Custom services let you add request/response behaviors outside the built-in UDS/KWP service handlers.

Use this when you need to emulate vendor-specific bootloader flows, tool handshakes, or test-only commands.

Overview

Each custom service defines:

- request bytes to match
- match strategy (strict, prefix, suffix)
- reply behavior:
 - default positive response
 - fixed response
 - multiple responses (responses) — send several replies to a single request, each with its own relative delay
 - replay sequence (replay + optional replayLoop)
- optional actions (same action model as security-level actions)

Custom services are checked before built-in service handlers.

Editor Workflow

1. Open the ECU's UDS or KWP workspace and the **Services** tab
2. Click **Add**; the new row is selected and its editor opens beside the table
3. Set **Name**, **Match Type**, and **Request Bytes**
4. Choose **Response Mode**:
 - **Default Positive Response**
 - **Fixed Response**
 - **Multiple Responses**
 - **Replay Sequence**
5. Optionally add **Actions**, **Delay**, or **Swallow Chance**

JSON Shape

```
{
  "name": "Transfer Data",
  "matchType": "prefix",
  "request": "B142",
  "delay": 5,
  "response": "F2 01",
  "actions": [
    {
      "kind": "overrideReplyId",
      "what": { "reply": "0x18DAF900" }
    }
  ]
}
```

Fields

Field	Type	Required	Notes
name	string	no	Label used in UI
matchType	string	yes	strict, prefix, or suffix
request	hex bytes	yes	Hex string or byte array in JSON
response	hex bytes	no	Fixed response payload
responses	array	no	Multiple responses sent for a single request
responses[].response	hex bytes	yes	Response payload
responses[].delay	number	no	Delay in ms relative to previous response (first is absolute)
replay	array	no	Sequence of responses with per-step count
replay[].count	number	yes (if replay item exists)	Number of hits for that step
replay[].response	hex bytes	yes (if replay item exists)	Response for that step
replayLoop	boolean	no	Restart replay sequence after the last step
delay	number	no	Delay in milliseconds before replying (not used with responses)
actions	array	no	Action model shared with security actions
debugSwallowResponseLikelihood	number	no	0.0–1.0 response drop probability

Response Modes

Default Positive Response

If none of response, responses, or replay is set, ECUmulator replies with a generated positive response.

Example for request B1 00:

- request SID 0xB1 becomes 0xF1
- second byte is echoed when present

Fixed Response

```
{
  "name": "VehiCAL Init",
  "matchType": "strict",
  "request": "3E 00 41 54 54 41",
  "response": "7E 41 54 41 54 40 80 89 09 0B 20 19 51 19 E4 3E 21 00 00 00 00 01"
}
```

Multiple Responses

Use responses when a single request should trigger several reply messages. Each entry has its own response bytes and a delay relative to the previous response (the first entry's delay is relative to receipt of the request).

```
{
  "name": "Multi-part reply",
  "matchType": "strict",
  "request": "36 01",
  "responses": [
    { "response": "76 01", "delay": 0 },
    { "response": "76 02 AA BB", "delay": 50 },
    { "response": "76 03 CC DD EE", "delay": 100 }
  ]
}
```

The service-level delay field is ignored when responses is set — use per-response delays instead.

Replay Sequence

Use replay when you want request N to return changing values over time.

```
{
  "name": "ReadFastArray",
  "matchType": "strict",
  "request": "3E 33 50 03 B0 7C",
  "replay": [
    { "count": 1, "response": "7E BA 09 54 16 20 20" },
    { "count": 3, "response": "7E B7 09 54 16 20 20" },
    { "count": 1, "response": "7E B5 09 54 16 20 20" }
  ],
  "replayLoop": true
}
```

Note: sequence behavior for custom services is represented by replay; there is no separate sequence field for customServices.

Match Types

- strict: request bytes must be exactly equal
- prefix: request must start with the configured bytes

- **suffix:** request must end with the configured bytes

prefix is common for block-transfer commands where payload length varies.

Actions

Supported action kinds:

- `setArbitration`
- `overrideReplyId`
- `lockAllOtherECUs`
- `unlockAllOtherECUs`
- `host`

Actions are optional and run when the custom service matches.

Internal Bus

The Internal Bus (Virtual CAN) service provides a software-based CAN bus that connects all configured ECUs and services without requiring physical hardware.

Overview

Internal Bus is the foundation of ECUmulator's architecture. It provides:

- **Shared message bus** for all ECUs
- **Protocol-independent routing** based on arbitration IDs
- **No hardware required** — runs entirely in software

How It Works

Virtual CAN Bus			
ECU 1 (0x7E0)	ECU 2 (0x7E1)	ECU 3 (0x7E2)	Services HSFZ, DoIP, ELM327, etc.

Each ECU listens for messages matching its configured Request ID and responds on its Reply ID.

Configuration

Internal Bus is automatically configured when you add ECUs to your vehicle:

Setting	Description
Bitrate	500 kbps (cosmetic, no actual timing)
Timing Factor	0 (instant delivery)

Requirements

The Internal Bus service must be enabled for any other service to function. It is the prerequisite for:

- HSFZ
- DoIP
- ELM327
- External CAN
- ECUconnect
- Logger

Transport-profile compatibility:

- `isotp` (classic): all listed services are available.
- `isotp_fd` (CAN FD + ISO-TP/FD): HSFZ, DoIP, and ELM327 are unavailable; ECUconnect, Internal/External CAN, and Logger remain available.

Message Flow

1. Service receives diagnostic request
2. Request is placed on virtual CAN bus
3. ECU with matching Request ID receives message
4. ECU processes and generates response
5. Response placed on bus with Reply ID
6. Service receives response and forwards to client

Runtime ECU State

When the emulator is running, each loaded ECU has live internal-bus state:

- **Enabled:** both RX and TX are armed on the virtual bus.
- **Disabled:** the ECU no longer receives requests and does not transmit responses on the virtual bus.

Studio and `ecp` both control this through the Remote Control API. The state is runtime-only and is reset by loading or applying a vehicle again; it does not change the saved ECU specification.

CAN Message Display

The **CAN Messages** panel shows recent frames on the virtual bus:

- **Green IDs** — Standard 11-bit CAN IDs
- **Orange IDs** — Extended 29-bit CAN IDs
- **[DLC]** — Data Length Code (classic 0–8, CAN FD up to 64 depending on transport profile)
- **Payload** — Hex bytes of the message

Performance

Internal Bus operates with negligible latency since no actual timing or arbitration occurs. For realistic timing simulation, use the timing factor setting (applies only to virtual mode).

External CAN

The External CAN service connects ECUlator to physical CAN interfaces so emulated ECUs can communicate on real CAN buses.

Overview

When enabled, ECUlator bridges the virtual CAN bus to a physical CAN adapter, allowing:

- **Hardware-in-the-loop testing** with real ECUs
- **Physical tool validation** using actual CAN hardware
- **Network simulation** with mixed virtual and physical nodes

Requirements

- **Linux:** SocketCAN interface available and configured.
- **macOS:** gs_usb-compatible, PEAK USB FD, TouCAN, or Tactrix OpenPort serial adapter available.
- **Windows:** J2534 PassThru driver installed for the selected adapter.
- **CAN interface** present and operational before enabling.

Supported Interfaces

Type	Examples
Linux SocketCAN	can0, vcan0, vcan1
macOS gs_usb	CANable/candleLight-compatible adapters (gs_usb:<vid>:<pid>:<address> with ECUlator++)
macOS PEAK USB FD	PCAN-USB FD / PCAN-USB Pro FD (peak_usb_fd:<pid>:<bus>:<address>:<channel>); channel 0 is CAN 1
macOS TouCAN	Rusoku TouCAN channels (toucan:0, toucan:1, ...)
macOS OpenPort	Tactrix OpenPort USB serial nodes (openport:/dev/cu.usbmodem...)
Windows J2534	PassThru adapters exposed via installed J2534 DLLs

Setup

Linux: Create Virtual CAN Interface (Testing)

```
sudo modprobe vcan
sudo ip link add dev vcan0 type vcan
sudo ip link set up vcan0
```

Linux: Configure Physical Interface

```
sudo ip link set can0 type can bitrate 500000
sudo ip link set up can0
```

Studio obtains macOS interfaces and capabilities from the selected runtime. A legacy `gs_usb:0` selection is migrated automatically when exactly one native `gs_usb` adapter is connected. For other legacy indices or multiple adapters, select the adapter in **Service Settings** → **External CAN** before enabling the service.

macOS: Tactrix OpenPort

OpenPort devices expose a USB CDC serial node such as `/dev/cu.usbmodemTAhJALxt1`. Use `openport:/dev/cu.usbmodem...` as the External CAN interface. Classic CAN at 500 kbit/s is supported; standard and extended 29-bit CAN IDs are bridged.

Using External CAN

1. Ensure your CAN interface is configured and UP
2. Enable **External CAN** in the Services panel
3. Open **Service Settings** → **External CAN**
4. Select your interface
5. Choose CAN mode:
 - **Classic CAN**: nominal bitrate preset
 - **CAN-FD**: nominal bitrate preset + data bitrate preset
6. Choose bus access mode:
 - **Normal**: read and transmit
 - **Listen-only**: receive only (no transmitted frames from ECUlator)
7. Apply settings (if External CAN is already enabled, the bridge is relaunched automatically)

Service toggles are applied through the runtime Remote Control API when a compatible runtime is already loaded. If the editor has unapplied vehicle changes or the runtime revision changed elsewhere, Studio reapplies the full bundle instead.

Interface Status

The dropdown shows available interfaces/channels with their status:

- **(available)** — Interface/channel is active and ready
- **(unavailable)** — Interface exists but is not currently usable

CAN controller health

By default, the External CAN card and bus canvas show the service connection. The checkbox selects the configuration; the badge shows the runtime state. **Pending** means the selected configuration is not active yet. A start failure or reconnection is shown directly, rather than being reduced to disconnected. The runtime header says **Ready**, **Active** or **Partly active** according to the services; a loaded vehicle alone does not mean those services have started.

To inspect the controller, open **Service Settings** → **External CAN** and enable **Show advanced CAN information**. This checkbox is off by default. It takes effect immediately, is saved in Studio's local preferences and does not reopen the adapter or modify the vehicle spec. Current CAN faults remain visible as a CAN error or warning with this checkbox off. A connected USB or TCP adapter can still be **error-passive** or **bus-off**. CANcorder shows the same health information in its Connection panel; `ecp list` and `ecp tui` report it for the runtime's External CAN service.

With advanced information enabled, CAN reception is shown separately. The service-row badge shows the controller state; the compact line below it shows reception, TEC/REC when available and nonzero error counters. Before the first received frame, reception remains **Unconfirmed**, even if the controller is error-active with zero error counters. A quiet bus and an adapter that is not connected to CAN cannot be distinguished yet. Check the bus connection, selected channel, bitrate and active peer. Once frames arrive, the card shows active reception or how long ago the last frame was received.

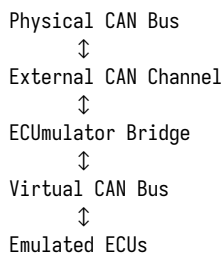
State	What to check
error-active	Controller can participate normally; check actual requests and replies too
error-warning	Error counters have reached the controller's warning threshold
error-passive	Controller has accumulated enough errors to send passive error flags
bus-off	Controller has withdrawn from the bus; correct wiring, bitrate and ACK conditions before recovery
stopped, sleeping	Controller is not operating normally on the bus
State unknown / unavailable	Driver or firmware has not supplied a current state
Stale	Last sample is retained after a status or connection failure

Available details include transmit/receive error counters (TEC/REC), observed error reports, ACK and protocol errors, bus-off transitions and queue overflows or drops. Unavailable counters are omitted rather than shown as zero. Report counts describe this adapter session; firmware counters may cover the device's longer running session. They are not a count of all errors across every node.

SocketCAN reads controller state and available counters through Linux; enable driver bus-error reporting when supported if detailed error reports are needed. Virtual vcan has no physical controller health. gs_usb capabilities depend on adapter firmware. PEAK and TouCAN expose their supported status messages. Older gs_usb firmware may report only controller state changes. After opening the adapter or reapplying a spec, its badge can remain **Unknown** on a quiet bus, even when ordinary frames establish reception. This means no current controller state has been reported yet. It does not mean the adapter lacks status support. A successful USB write alone does not establish a CAN ACK. ECUconnect's updated firmware reads the MCP2518FD controller directly; remote adapter clients need that firmware's health RPC. Older ECUconnect firmware, OpenPort, J2534 and the logger-only TCP stream report unavailable information.

This is not a termination measurement. For a two-node bench, use a 120 Ω resistor between CAN-H and CAN-L at each end and a common reference ground. With power off, the two terminators in parallel measure about 60 Ω . Match bitrate and CAN mode, and use normal access mode on the node that must ACK; listen-only nodes do not acknowledge frames. error-active alone does not prove correct wiring, termination or communication.

Traffic Flow



Platform Support

Platform	Status
Linux	SocketCAN (classic + CAN-FD)
macOS	gs_usb and PEAK USB FD (classic + CAN-FD), TouCAN and OpenPort serial (classic CAN)
Windows	J2534 (classic CAN)

On unsupported platforms, the External CAN option is hidden.

Troubleshooting

Issue	Solution
No interfaces shown	Connect a supported adapter; on Linux create/configure interface; on macOS verify the adapter driver/device node; on Windows verify the J2534 driver is installed
Interface down	Linux: run <code>sudo ip link set up <interface></code>
Permission denied	Add user to dialout group or run with <code>sudo</code>
No traffic seen	Verify nominal bitrate and CAN mode match your bus
CAN-FD frames missing/failing	Ensure CAN-FD mode is selected and data bitrate matches the physical network

Diagnostic Connection Log

With ECUmulator++, the External CAN connection log includes semantic KWP, UDS, and OBD2 requests and responses from the emulated ECUs. Messages are recorded at the application boundary after request reassembly and before response segmentation, including KWP over TP2.0. Flow-control and transport acknowledgements remain in the CAN message table.

Each diagnostic entry identifies the ECU, direction, service, and current KWP/UDS session. Negative responses include their NRC, including Response Pending. Suppressed responses do not produce a response entry. Payload previews contain up to 64 bytes; longer messages show their full length. The log describes ECU processing and generated responses, not confirmation of physical delivery.

HSFZ

HSFZ (High Speed Fahrzeug-Zugang) is a BMW proprietary Ethernet-based diagnostic protocol used for high-speed communication with vehicle ECUs.

Overview

The HSFZ service exposes a TCP server that accepts diagnostic tool connections and routes requests to the configured ECUs.

Transport compatibility:

- Requires classic transport profile (isotp).
- Unavailable when **Vehicle** → **Transport Profile** is set to isotp_fd.

Default Configuration

Setting	Value
TCP Port	6801
UDP Port	6811 (discovery)
Protocol	ISO-TP over Ethernet

Features

- **TCP diagnostic communication** — Full request/response handling
- **UDP discovery** — Responds to vehicle identification broadcasts
- **ISO-TP framing** — Automatic segmentation for multi-frame messages
- **Session management** — Tracks diagnostic sessions per ECU

Connection

Connect to the HSFZ service using:

- BMW ISTA/D diagnostic software
- Compatible OBD tools with HSFZ support
- Custom TCP clients on port 6801

Loopback tip: use 127.0.0.1 instead of localhost for local connections (localhost can resolve to IPv6 ::1 while HSFZ listens on IPv4).

Protocol Flow

1. Client connects to TCP port 6801
2. Client sends diagnostic request with target ECU address
3. ECUmulator routes request to matching ECU
4. ECU processes request and generates response

5. Response is sent back to client

Limitations

- Single client connection at a time
- No TLS/encryption support
- UDP discovery uses broadcast (may not work across subnets)

Troubleshooting

Issue	Solution
Connection refused	Ensure HSFZ service is enabled and Internal Bus is active
No UDP discovery response	Check firewall settings for UDP port 6811
Request timeout	Verify ECU arbitration IDs match your diagnostic tool configuration

DoIP

DoIP (Diagnostics over Internet Protocol) is a standardized Ethernet-based diagnostic protocol defined in ISO 13400.

Overview

The DoIP service provides TCP/UDP communication for diagnostic tools, following the ISO 13400 standard for vehicle identification and diagnostic messaging.

Transport compatibility:

- Requires classic transport profile (isotp).
- Unavailable when **Vehicle** → **Transport Profile** is set to isotp_fd.

Default Configuration

Setting	Value
TCP Port	13400
UDP Port	13400 (discovery)
Protocol	ISO 13400-2

Features

- **Vehicle identification** — UDP broadcast discovery with VIN response
- **Routing activation** — Standard DoIP activation handshake
- **Diagnostic messaging** — Full UDS over DoIP support
- **Entity identification** — Responds with configured vehicle information

Connection Flow

1. **Discovery** — Client broadcasts UDP vehicle identification request
2. **Response** — ECUmulator responds with VIN and entity address
3. **TCP Connect** — Client establishes TCP connection
4. **Routing Activation** — Client sends activation request
5. **Diagnostic Session** — Normal UDS communication begins

Loopback tip: for local testing, connect to 127.0.0.1 rather than localhost (localhost may resolve to IPv6 ::1 while DoIP listens on IPv4).

Supported Payloads

Payload Type	Support
Vehicle Identification	Yes
Routing Activation	Yes
Diagnostic Message	Yes
Alive Check	No
Entity Status	No

VIN Configuration

The VIN returned in DoIP discovery responses is configured in the Vehicle panel. A valid 17-character VIN is recommended for compatibility with diagnostic tools.

Troubleshooting

Issue	Solution
No discovery response	Check UDP port 13400 is not blocked
Routing activation failed	Ensure Internal Bus service is enabled
Diagnostic timeout	Verify ECU arbitration IDs match your tool

ELM327/STN

The ELM327/STN service provides a TCP interface that emulates an ELM327 or STN2xxx OBD-II adapter, allowing standard OBD-II diagnostic applications to communicate with the emulated ECUs.

Overview

This service exposes a TCP server that accepts AT/ST commands and translates them into CAN bus requests for the configured ECUs.

Transport compatibility:

- Requires a non-FD transport profile (isotp or sae_tp20).
- Unavailable when **Vehicle** → **Transport Profile** is set to isotp_fd.

Default Configuration

Setting	Value
TCP Port	35000
Default Protocol	Auto (ISO 15765-4 CAN)

Studio can also override reported identity/ignition/voltage/manufacture strings for ATI, AT@1, AT@2, ATIGN, ATRV, STI, STIX, STSN, and STMFR. Studio can optionally write full per-connection ELM TCP transcripts (RX/TX + commands) when **Log Directory** points to a valid path.

Supported Commands

AT Commands (ELM327)

Command	Description
ATZ	Reset
ATE0/ATE1	Echo off/on
ATH0/ATH1	Headers off/on
ATSP x	Set protocol
ATTP x	Try protocol
ATDP	Describe protocol
ATIGN	Read ignition status
ATSH xxx	Set header
ATCF xxx	Set CAN filter
ATCM xxx	Set CAN mask

Command	Description
ATST xx	Set timeout

ST Commands (STN2xxx)

Command	Description
STI	Print firmware ID
STIX	Print extended firmware ID
STSN	Print serial number
STMFR	Print manufacturer
STDI	Print device ID
STCSEGR0/1	Auto-segmentation RX off/on
STCSEGT0/1	Auto-segmentation TX off/on
STPX...	Set modifiers for next bus command
STPPMA...	Add periodic CAN message
STPPMD...	Delete periodic message
STPPMC	Clear periodic messages

TP2.0 Commands

When **Vehicle** → **Transport Profile** is sae_tp20, the adapter also exposes TP2.0 channel commands:

Command	Description
TPOPEN[, <target>...]	Open a TP2.0 channel. <target> defaults to 01.
TPSTAT	Show active TP2.0 channel state.
TPCLOSE	Close the active TP2.0 channel.

After TPOPEN, normal hex requests such as 1A90 are sent as KWP-over-TP2.0 automatically.

OBD-II Mode Support

Mode	Description	Support
01	Current Data	Yes
02	Freeze Frame	Yes
03	Stored DTCs	Yes
04	Clear DTCs	Yes
05	O2 Monitoring	Yes
06	On-Board Tests	Yes

Mode	Description	Support
07	Pending DTCs	Yes
08	Control Operations	Yes
09	Vehicle Info	Yes
0A	Permanent DTCs	Yes

Connection

Use any ELM327-compatible application (Torque, OBD Fusion, custom scripts) and connect via TCP:

Host: 127.0.0.1 (or machine IP)

Port: 35000

Loopback tip: avoid localhost here (localhost can resolve to IPv6 ::1 while the adapter listens on IPv4).

Protocol Selection

The adapter auto-detects the CAN protocol based on ECU configuration:

- **CAN 11-bit 500kbps** — Standard OBD-II
- **CAN 29-bit 500kbps** — Extended addressing (J1939-style)

Troubleshooting

Issue	Solution
UNABLE TO CONNECT	Verify ECU has OBD-II protocol configured
NO DATA	Check PID is defined in ECU configuration
Connection refused	Ensure ELM327 service is enabled

For TP2.0:

- UNABLE TO CONNECT usually means the loaded vehicle is not using sae_tp20, or the target ECU lacks transport.tp20 endpoint metadata.
- Use TPSTAT after TPOPEN to confirm the negotiated target and channel IDs.

ECUconnect

ECUconnect provides a CANyoneo-compatible TCP adapter endpoint for diagnostics and firmware-update flows.

This is the emulator *offering* that endpoint to a diagnostic app, which is the opposite direction from using a physical ECUconnect adapter as a CAN interface. The two use different ports: the emulator serves on 8129, a real adapter serves on 129.

ECUconnect as a remote runtime

When Studio controls ECUlator++ running on a physical ECUconnect, choose that device as the network runtime. Its onboard **K-Line** service is described in [K-Line on ECUconnect](#). The TCP adapter emulation settings below do not enable the physical K-Line responder.

Default Configuration

Setting	Value
TCP Port	8129
Model	ECUconnect
Default Version	0.9.999
Default Hardware Revision	PVIRT

Settings

In **Settings** → **Service Configuration** → **ECUconnect**, you can configure:

- **Version Number** (reported by request_info)
- **Hardware Revision** (reported by request_info)

When ECUconnect is enabled, changing either value restarts only the ECUconnect service.

Loopback tip: when connecting from the same machine, use 127.0.0.1 instead of localhost (localhost may resolve to IPv6 ::1 while the service listens on IPv4).

Virtual Firmware Update Flow

Supported update PDUs:

- prepare_for_update
- send_update_data
- commit_update
- reset

Timing simulation:

- Prepare delay: **3 seconds**
- Write delay: **50 ms per chunk**

- Commit delay: **2 seconds**

The service replies with ok for each update stage. After `commit_update`, the uploaded binary is scanned for a firmware version string. After reset, the connection is dropped and ECUconnect restarts with the new reported firmware version.

Persistence + Update Note

If the version came from a virtual firmware update:

- Studio marks it as update-derived
- Studio stores when the update happened (UTC timestamp)
- The Settings dialog shows a note with that timestamp

This metadata persists in Studio autosave state and survives Studio restarts.

Limitations

- Legacy update-status PDUs are not emitted (`update_started_send_data`, `update_data_received`, `update_completed`)
- RPC PDUs are not implemented
- Receive PDUs may be compressed when compression reduces wire size.

Transport Protocols

- `open_channel` supports classic raw (0x00) and isotp (0x01) channels.
- The ECUmulator++ desktop service also supports tp20 (0x07) channels after the usual fixed-ID TP2.0 setup was negotiated over a raw channel.
- `open_fd_channel` supports raw_fd (0x03) and isotp_fd (0x04) channels.
- Raw FD channels accept payloads up to 64 bytes.
- FD channels are intended for vehicle bus transport profile isotp_fd (Transport Profile dropdown in Vehicle panel).

ENET Service Broker

ENET Service Broker connects ECUmulator to a remote ESB instance as the **vehicle** endpoint for a VIN-scoped session.

Overview

When enabled, ECUmulator:

1. Discovers `enet-service-broker._enetbroker._tcp` via Zeroconf when no URL override is set (or uses explicit URL override).
2. Calls `POST /port/:vin` on the discovered/overridden broker URL.
3. Receives a vehicle port and matching tester port (`vehicle + 1000`).
4. Connects to the returned vehicle TCP port.

5. Forwards CAN traffic bidirectionally over the CANvoy TCP packet format in the selected transport mode.
6. You can choose transport mode in Settings: isotp (ISO-TP PDUs, default) or raw (CAN frames).

Requirements

- A complete 17-character VIN is required.
- Internal Bus must be enabled (auto-enabled when you toggle this service).
- Zeroconf discovery must be available for auto mode (empty URL override), or a reachable URL override must be provided.

Loopback tip: use 127.0.0.1 instead of localhost for local broker testing.

Default Configuration

Setting	Value
Broker URL Override	empty (auto-discover via Zeroconf)
Role	Vehicle
Transport Mode	isotp (default) or raw

Tester Port Display

The Services panel shows the allocated **Tester Port** below the ENET service badge so you can share it with the tester operator.

Troubleshooting

Issue	Solution
Service fails to start	Ensure VIN is present and 17 characters long
No session allocation	Verify Zeroconf visibility on LAN or set a valid Broker URL override
Frequent reconnects	Check broker health and tester connectivity

K-Line on ECUconnect

Studio can control ECUlator++ running directly on ECUconnect over Wi-Fi. The device answers a physical K-Line tester locally, including initialization and response timing. Studio edits and uploads the vehicle configuration.

Start with a demo

1. Select your ECUconnect network runtime in Settings. It must run the standalone ECUlator++ firmware with the onboard K-Line responder.
2. Open **Demo Vehicles** → **K-Line** → **AL319 OBD-II** or **AL319 OBD-II · 2 ECUs**.
3. In the vehicle panel, the **K-Line** line under **Transport profile** sums up the binding; **Set up...** opens it. Check the ECU assignments and addresses.
4. Enable the **K-Line** service on the remote runtime with interface onboard and bus diagnostic, and apply the vehicle.
5. Start an OBD connection on the tester. On a mixed bench, disable other CAN responders if you want the tester to select K-Line.

An older open demo is not automatically replaced when its bundled file changes. Reload it from the demo menu to get new fixture data; this also resets your unsaved demo edits. The demo speed starts at 0 km/h.

Multiple ECUs on one line

A vehicle can bind 1–16 named ECUs to distinct physical K-Line addresses. The two-ECU demo uses Engine at 0x10 and Transmission at 0x18, tester 0xF1, and functional initialization address 0x33.

A physical diagnostic request is routed to its addressed ECU. A functional request can receive replies from both ECUs, each carrying its own source address. Initialization belongs to the shared physical line; each ECU has its own diagnostic data and session state. ECU names and addresses must be unique. Each bound ECU needs an OBD-II or KWP handler.

Supported profile

The current onboard profile supports KWP2000 Fast Init and 5-baud Slow Init, automatic selection between them, 10400 baud and keywords E9 8F. Slow Init support does not imply an ISO 9141-2 diagnostic implementation. The AL319 may probe several protocols before choosing KWP2000.

Vehicle configuration owns the logical bus, ECU/tester addresses, initialization mode and response delay. Hardware pins and UART selection belong to firmware. A desktop runtime without the onboard backend cannot provide this physical service merely by loading a K-Line vehicle. Studio then lists the K-Line service as not available, without status or setup hints, and shows the vehicle's **K-Line** line only when the open vehicle already has a binding.

Diagnostic data

Use the existing OBD-II and KWP workspaces to edit the bound ECU's data. The AL319 demos supply readiness, coolant, RPM, speed, VIN, Calibration ID, CVN and ECU name. VIN, Calibration ID and CVN have been confirmed on the AL319; ECU-name display depends on the tester and has not been confirmed.

The two-ECU demo additionally has:

ECU	Stored codes	Pending codes	Permanent codes
Engine	P0300, P0171, P0420	P0133	P0420
Transmission	P0715, P0741	P0720	P0741

MIL is on, with three stored codes for Engine and two for Transmission. Stored and pending DTC display passed the AL319 bench check. The AL319 manual (section 4.1) limits permanent-code reading to CAN; our permanent-code responses are covered by software checks. Readiness bytes are configured data: keep them consistent when changing the fault memory.

Engine also provides two small Mode 02 freeze frames:

Frame index	Associated DTC	Coolant	RPM	Speed
00	P0171	85 °C	2250 rpm	60 km/h
01	P0420	95 °C	3000 rpm	100 km/h

These are configured snapshots. Frame indices are independent of the order of DTCs in the fault list. The AL319 displayed frame 00 correctly and requested only that index in the bench trace. Its manual describes scrolling through measurements, with no frame-selection step. Frame 01 also passes a direct shared-runtime check; its physical K-Line exchange remains untested.

Editing while connected

Changing diagnostic values and applying the vehicle preserves the physical K-Line connection when the binding and line settings stay compatible. Queued replies finish with their original data; following requests use the replacement. Diagnostic session and sequence state comes from the newly applied spec. Live speed changes have passed repeated physical AL319 checks while Studio status polling and device persistence remain active.

Inline constant editing keeps your draft and caret while live previews update. Enter or leaving the cell commits, Escape cancels, and Tab commits and moves to another editable value. Apply the spec to send edits to the remote runtime.

Changing line settings or assignments, stopping the runtime or disabling the service requires another tester initialization. ECU arming changes currently restart the K-Line backend too. Reopening Studio is a control connection; load/apply actions and service changes determine whether the physical line is reconfigured.

Status and LED

K-Line service status shows whether the device is listening or connected, traffic counts, link errors and a bounded list of recent events. This is a separate diagnostic trace from CAN frame logging. Save a longer capture when investigating an intermittent problem; recent events eventually roll out.

The red ECUconnect LED is intended to stay on while connected, pulse dark for traffic, and turn off on disconnect or service stop. Firmware support is implemented; a visual bench check remains pending.

After flashing or rebooting ECUconnect, start a fresh connection from the scanner's main menu. Old tester traffic can reach the initialization listener before a new handshake and produce Capture errors. Error 13 alone does not identify the cause: retain its init reason and pulse durations. A successful subsequent connection should be assessed separately from these pre-init events.

Validation scope

The physical AL319 checks cover two ECU identifiers, readiness, live data, VIN/CALID/CVN, stored/pending DTCs, long request runs and repeated live speed updates. The first freeze frame also passed. Further checks remain for LED appearance, Studio arming/toggle/reconnect flows, RPM/VIN edits, concurrent CAN load and broader KWP services with delayed responses.

Reference: [AL319 manual, sections 4.1 and 4.4](#).

Writing Vehicle and ECU Specs

After [Your First Emulation](#), use this advanced tutorial when you want to author the files directly. You already know the edit, apply and retest cycle; here you learn how the saved model expresses it.

This guide takes you from three editable JSON files to one simulated ECU with UDS and OBD-II handlers. It describes the shipped native runtime and Studio; no Python installation is required. The demo is educational, not a reproduction of a real vehicle. Its VIN is a fictitious 17-character test string.

1. Start with the included demo

Choose **Demo Vehicles** → **BMW Motorrad — 1250 Boxer**. The menu label may also include the file path. Expect one logical ECU, **BMS-X**, with two diagnostic handlers.

File	Purpose
bmw-motorrad-vehicle.json	Vehicle name, VIN and references to both handlers
bmw-motorrad-uds.json	UDS addresses, timing and identification DIDs
bmw-motorrad-obd2.json	OBD-II addresses, 24 live PIDs, freeze-frame data and VIN

The macOS installer installs reference files under **/usr/local/cansole/share/specs/motorrad/**. Copy the whole directory before editing:

```
mkdir -p "$HOME/Documents/ECUlator"
cp -R /usr/local/cansole/share/specs/motorrad "$HOME/Documents/ECUlator/motorrad"
```

Keep all three files together and open your copied vehicle file in Studio. Reload after making external text edits. The same demo is bundled inside the application for offline use; do not edit files inside the application bundle. [Example Spec Files](#) contains the complete files if you need to recreate them.

For other protocol, address and hardware combinations, see [Demo Vehicles](#).

2. Vehicle, ECU and handler

A vehicle contains named ECU entries. Each source file describes one diagnostic protocol handler. Entries with the same vehicle-level **name** belong to one logical ECU:

```
{
  "name": "BMW Motorrad — 1250 Boxer",
  "info": "Educational BMW Motorrad BMS-X example for building and testing a diagnostic client against one physical ECU with both UDS and OBD-II. Curated by Vanille Media.\n\nEmulates identification DIDs, live and freeze-frame OBD-II data, readiness and monitor results, VIN information, stored and permanent DTCs, and values that change through generators or sequences. Both handlers share CAN request 0x7E0 and reply 0x7E8; OBD-II also listens on 0x7DF. The authoring guide uses this vehicle step by step.\n\nUses a synthetic VIN and illustrative values. It does not reproduce the complete behavior of a real motorcycle or implement its flash programming and security algorithms.",
  "vin": "WB10G4105T6D00001",
  "ecus": [
    {
      "name": "BMS-X",
      "path": "bmw-motorrad-uds.json"
    },
    {
      "name": "BMS-X",
      "path": "bmw-motorrad-obd2.json"
    }
  ]
}
```

Both entries deliberately use BMS-X. A different name describes a different logical ECU. The name inside each ECU source file is a descriptive label.

The optional vehicle-level **info** field contains the author's description of the specification: its purpose, the functions and protocols it emulates, and its limitations. Plain text and Markdown are supported. Use the **Info** button beside **Vehicle** to open the Markdown editor with formatting tools and a live preview. **Apply** updates the vehicle; **Cancel** discards the draft. Save the vehicle file to keep the changes. The description is included in runtime bundles, so it also survives reconnecting to a running emulator. Every shipped demo includes this description. Older files may omit it.

Paths are relative to the **vehicle file**, not the terminal directory. Studio and **ecp load** resolve the files and upload their contents as a bundle; the target does not need access to your original paths. Loading an individual ECU file works too, but loads only that handler.

Inline vehicle formats are also supported. This guide uses separate files for easy reuse; Studio saves or exports may use a resolved representation instead.

The vehicle VIN and protocol responses are separate data. Keep the vehicle VIN, UDS DID F190, the OBD-II vin field and its Mode 09 VIN bytes consistent. Changing one field does not guarantee the others change.

3. JSON rules and comments

Use UTF-8 **strict JSON**. JSON5 is not supported: no `//` or `/* */` comments, trailing commas, single quotes, or unquoted hexadecimal literals. Quote hexadecimal IDs as strings:

```
{ "request": "0x7E0", "reply": "0x7E8" }
```

PID objects can contain ordinary **comment** string fields, as in our examples:

```
{
  "id": "0x0C",
  "comment": "Engine speed: raw value divided by 4 = 2000 rpm.",
  "simple": "1F 40"
}
```

These fields explain the data and do not affect runtime responses. They are not JSON comment syntax. Do not add unknown fields indiscriminately: some schema objects reject them. An editor/export operation may also omit fields it does not model. Keep your annotated source files and this guide together.

4. Addresses and diagnostic payloads

Both handlers use physical request ID **0x7E0** and reply ID **0x7E8**. The OBD-II handler additionally accepts functional request ID **0x7DF**. The UDS auxiliary list is empty in this example.

Field	Meaning
request	CAN arbitration ID sent by the tester
reply	CAN arbitration ID emitted by the ECU
auxiliary	Additional request IDs, such as functional requests

The handlers share addresses but own different services: OBD-II Modes 01–0A and UDS services such as 22. They form one ECU rather than independent responders that reject each other's requests. If you add another ECU, choose its physical address pair deliberately; renaming it alone does not change its address.

Spec values contain data, not CAN/ISO-TP headers:

- Configured engine-speed bytes: **1F 40**.
- Diagnostic response: **41 0C 1F 40**.
- Classic CAN single frame: **04 41 0C 1F 40 00 00 00**.

The first CAN byte is the payload length. The runtime adds the service/PID prefix and ISO-TP framing. Long responses, such as VINs, require flow control from the tester. This demo uses Classic CAN, not CAN-FD.

5. UDS identification DIDs

A UDS file has "kind": "uds" and a matching **uds** object. Its identifiers array defines ReadDataByIdentifier (service 22) responses:

```
{
  "id": "0xF187",
  "name": "Part Number",
  "type": "ascii",
  "content": "13618555123"
}
```

Request payload **22 F1 87** returns **62 F1 87** followed by the ASCII content. Do not include that response prefix in content. The demo supplies VIN (F190), part number (F187), hardware version (F191) and software version (F195). Use distinct DID IDs when adding entries.

The complete demo contains 16 DIDs: ten text identifiers and six binary measurements/status values. F100–F105 are demo-defined, not standardized measurement definitions. F100 (rpm), F101 (coolant), F102 (vehicle speed) and F103 (voltage) mirror the corresponding OBD-II values.

For binary content, use a bytes field rather than putting hex text in content:

```
{
  "id": "0xF100",
  "name": "Engine Speed",
  "bytes": "1F 40"
}
```

Request 22 F1 00 returns 62 F1 00 1F 40. Here "content": "1F 40" would encode the text characters instead of those two bytes. Use either content or bytes.

An identifier can also answer with a sequence, one step per read, like OBD-II PID sequences (the same for KWP identifiers):

```
{
  "id": "0xF1A0",
  "name": "Counter",
  "bytes": "00",
  "sequence": { "responses": ["01", "02", "03"], "repeat": "loop" }
}
```

Three reads of 22 F1 A0 answer 01, 02, 03, the fourth 01 again; with "repeat": "once" it stays on 03. A write (2E, UDS 2F short-term adjustment) pauses the sequence and the identifier returns what was written, until an ECU reset or applying the vehicle starts it over. The fixed value is kept for when the sequence is removed.

The timing fields p2ServerMax, p2extServerMax and s3SessionTimeout are in **seconds**. The demo uses 0.05, 2.0 and 5.0 respectively.

Sessions and the security that guards them

Without a sessions list the ECU enters every session it is asked for and answers each with the timing above. A list says what differs per session and what guards entering it:

```
{
  "sessions": [
    { "session": "0x03", "p2ServerMax": 0.02 },
    { "session": "0x02", "p2extServerMax": 5.0,
      "requiresSecurityLevels": ["0x11"], "reachableFrom": ["0x03"] }
  ],
  "rejectUnlistedSessions": true,
  "securityLevels": [
    { "level": "0x11", "sessions": ["0x03"],
      "seedKeyPairs": [{ "seed": "12 34", "key": "56 78" }] }
  ]
}
```

- A session's own P2/P2* go into its 50 reply; what it leaves out comes from the ECU.
- reachableFrom: 10 02 from any other session gets **7F 10 22**.
- requiresSecurityLevels: without one of these levels unlocked, **7F 10 33**.
- rejectUnlistedSessions: unlisted sessions get **7F 10 12**. The default session 01 is always entered.
- securityLevels[].sessions: 27 11 anywhere else gets **7F 27 7F**. A level without sessions keeps the old rule: anywhere but session 01.
- Switching to another session locks security again, except in a session whose requiresSecurityLevels the unlocked level satisfies.
- A level's delay (milliseconds) is how long checking the key takes: the ECU answers 7F 27 78 at P2, repeats it every pendingIntervalMs (block level, kept below P2*; without it, 100 ms before P2* of the active session runs out), and answers after the delay. The same applies to every delay/delayMs in a spec; KWP sends its first 78 at the block's p2ServerMax (default 0.05 s). While an answer is

pending, other requests get **7F xx 21**. keyTimeoutMs is how long a seed stays valid; a later key gets **7F 27 24** (KWP 7F 27 22) and needs a new seed.

KWP uses the same keys with its own session numbers (81 standard, 85 programming, 86 development, 87 adjustment; SSF 14230-3). Its 50 reply carries no timing: a session's timingParameters are what 83 answers while it is active, and 10 81 brings the ECU's back. responseData and delayMs on a session replace the older sessionProfiles, which are still read. A level outside its sessions gets **7F 27 80**, KWP's code, and "stopDiagnosticSession": false makes 20 answer **7F 20 11**, as on SSF 14230-3 ECUs.

Security levels, routines, memory layout and startup actions are empty. No seed/key prerequisite is needed for the identification examples. Extend these features only when your test scenario needs them; [Security Access](#), [Routines](#) and the other configuration chapters cover them. Labels explain a DID; the ID and content determine the actual response.

The UDS fault memory carries the same three faults the OBD-II handler reports, each with the status bits that decide which query returns it:

DTC	Status names	Byte
01 30 00 (P0130)	testFailed, confirmedDTC, warningIndicatorRequested	89
02 01 00 (P0201)	testFailed, confirmedDTC	09
01 17 00 (P0117)	testFailedThisOperationCycle, pendingDTC	06
15 20 13 (P1520)	testFailed, confirmedDTC	09

P1520 is the gear shift assistant sensor: manufacturer specific, not emissions relevant, and therefore present only here. It carries no warning indicator, because the MIL is an emissions lamp.

19 02 FF returns all four with their status byte, **19 01 FF** returns the count, and **14 FF FF FF** empties the memory. The status names map onto the bits in ISO 14229 order, so a name you misspell silently drops its bit.

6. OBD-II live PIDs

obd2.frames[0].pids supplies Mode 01 current data. Give every PID a unique ID within its frame and exactly one source: **simple**, **sequence** or **dynamic**.

A simple value is a hex byte string containing only PID data. For example, "32" means the single byte 0x32 (decimal 50), not the characters "3" and "2". In the table, A and B denote the first and second bytes.

PID	Example meaning	Bytes / decoding
01	Monitor status, MIL off	00 07 FF 00
03	Fuel system 1 closed loop	02 00
04	Load $\approx$ 50.2%	80; $A \times 100 / 255$
05	Coolant 90 °C	82; $A - 40$
06	Short-term trim 0%	80; $(A - 128) \times 100 / 128$
07	Long-term trim 0%	80; same encoding
0A	Fuel pressure 300 kPa	64; $A \times 3$

PID	Example meaning	Bytes / decoding
0B	Manifold pressure 100 kPa	64; A
0C	Engine speed 2000 rpm	1F 40; (256A + B) / 4
0D	Vehicle speed 50 km/h	32; A
0E	Timing advance 0°	80; A / 2 – 64
0F	Intake air 25 °C	41; A – 40
11	Throttle $\approx$ 25.1%	40; A $\times$ 100 / 255
14	O2 sensor 1: 0.45 V, no trim	5A 80; A / 200, B – 128
1C	EOBD compliance	06
1F	Run time 10, 20, 30 seconds	Sequence
21	Distance with MIL on: 0 km	00 00; 256A + B
2F	Tank level 25–75%	Dynamic
30	Warm-ups since cleared: 40	28; A
31	Distance since cleared: 8000 km	1F 40; 256A + B
33	Barometric pressure 100 kPa	64; A
34	O2 sensor 1 wide range, λ 1.0	80 00 80 00; (256A + B) / 32768
42	Module voltage 14 V	36 B0; (256A + B) / 1000
44	Commanded λ 1.0	80 00; (256A + B) / 32768
45	Relative throttle $\approx$ 10.2%	1A; A $\times$ 100 / 255
46	Ambient temperature 20 °C	3C; A – 40
49	Throttle grip sensor 1 $\approx$ 25.1%	40; A $\times$ 100 / 255
4A	Throttle grip sensor 2 $\approx$ 25.1%	40; redundant channel
4C	Commanded throttle $\approx$ 25.1%	40; A $\times$ 100 / 255
51	Gasoline fuel type	01
5C	Engine oil temperature 70 °C	6E; A – 40
5E	Engine fuel rate 4.0 L/h	00 50; (256A + B) / 20
F1	Custom lab integer: 4660	12 34; unsigned big-endian
F2	Custom lab text: DEMO	44 45 4D 4F; ASCII
F3	Custom lab status: 0, 1, 2	Sequence

Supported-PID bitmap requests (00, 20, etc.) are handled by the runtime; do not maintain bitmap bytes manually here. Monitor status and compliance fields are fixed sample values, not an automatic model of the engine or fault state.

Custom OBD-II PIDs

The runtime accepts custom PID entries in the same pids array. For example:

```
{
  "id": "0xF1",
  "comment": "Custom lab value: unsigned big-endian integer 4660.",
  "simple": "12 34"
}
```

Mode 01 request 01 F1 returns 41 F1 12 34. The runtime includes configured PIDs in its supported-PID bitmaps, including the E0 bitmap for these examples.

Rules:

- PID IDs are one byte: 00–FF. Use unique IDs within each frame.
- 00, 20, 40, 60, 80, A0, C0 and E0 are supported-PID bitmap requests; the runtime handles them, so do not use them for custom data.
- F1–F3 here are illustrative lab assignments, not a claim that these numbers are universally available or standardized. Agree on IDs with your tester.
- A generic OBD application may ignore unknown PIDs or have a conflicting interpretation. Configure its request, response length, encoding, unit and label explicitly.
- Custom PID data can use simple, sequence or dynamic just like other PIDs. This does not add arbitrary diagnostic services or two-byte PID IDs. For service 22 with a two-byte identifier, configure the UDS handler's DIDs.

Sequences change on reads

```
{
  "id": "0x1F",
  "sequence": {
    "responses": ["00 0A", "00 14", "00 1E"],
    "repeat": "loop"
  }
}
```

Successive reads return 10, 20 and 30 seconds, then repeat. This is a repeatable display test, not an elapsed-time counter. Other clients polling that PID advance the sequence too.

Generators change with time

```
{
  "id": "0x2F",
  "dynamic": {
    "function": "sawtooth",
    "min": 25,
    "max": 75,
    "period": 60,
    "encoding": { "type": "uint8", "scale": 2.55 }
  }
}
```

This generator varies a physical percentage over 60 seconds. Encoding performs the inverse of the tester's decoding: $\text{raw} = \text{percentage} \times 255 / 100$. Integer encoding quantizes the result. Sawtooth wraps at the end of the cycle. Use a curve for more elaborate patterns; see [Dynamic Values](#).

Do not assume a first read starts at the beginning of the cycle. Use static values for exact response checks.

7. Freeze frames and VIN

frames[0] holds live values. frames[1] supplies the first Mode 02 freeze frame (request index 00): coolant at 50 °C and engine speed at 1000 rpm. These are configured snapshot values, not an automatic capture when a DTC occurs.

The informational array supplies four Mode 09 values: VIN (02), calibration ID (04), a demonstration calibration verification number (06) and ECU name (0A). The VIN value contains a count byte (01) followed by its 17 ASCII bytes. The calibration ID has one 16-byte ASCII item; the CVN is four fixed sample bytes, not a checksum calculated from the ECU data. Keep the VIN consistent with the UDS VIN. The tutorial regression test compares all four VIN source representations and both diagnostic responses byte for byte.

Fault memory and monitor results

Which query returns a DTC follows from its status, not from the array it sits in. **confirmedDTC** puts it in mode 03, **pendingDTC** in mode 07, and the separate **permanentDtcs** array feeds mode 0A:

DTC	Reported by	Meaning
01 30 00	03, 0A	P0130 oxygen sensor circuit, confirmed, MIL requested
02 01 00	03	P0201 injector circuit cylinder 1, confirmed
01 17 00	07	P0117 coolant sensor low, pending only

04 empties the stored and pending memories. It does not touch the permanent one, which is the point of that array: ISO 15031-5 lets a permanent DTC clear only when its own monitor passes, so a tool cannot hide a live fault. Mode 01 PID 01 keeps its configured bytes across a clear, because monitor status is a fixed sample value rather than a model of the fault state.

The two handlers keep separate fault memories, and that models a real division rather than working around one. OBD-II is legally scoped to emissions relevant faults: mode 03, 07 and 0A expose only those, and mode 04 clears only those. Everything else a controller records — manufacturer specific faults, and engine faults without emissions relevance — lives in a deeper layer that a generic scan tool never sees. That layer is reached over UDS with **19** and cleared with **14 FF FF FF**.

So mode 04 not reaching the UDS memory is the behaviour to expect: clearing the emissions memory must not wipe the manufacturer fault memory, or a workshop would lose the fault history every time someone plugged in a generic reader. Getting at the deeper layer takes a manufacturer diagnostic tool, which is exactly the asymmetry the two handlers reproduce.

The bundle demonstrates that asymmetry rather than only describing it. Three emissions faults sit in both memories; P1520 sits only in the UDS one. A generic reader sees two stored faults and clears them with mode 04. A manufacturer tool sees four, and after that same mode 04 the gear shift fault is still there, because no OBD-II service may report or clear it.

onBoardMonitoring holds mode 06 test results, one record per monitor ID:

MID	Record	Meaning
01	01 0B 1F 40 2C D3 46 71	O2 B1S1 rich to lean: 0.244 V below its 0.350 V limit
02	02 0B 2E 1B 23 28 39 A2	O2 B1S1 lean to rich, inside its limits
21	81 01 01 90 00 00 02 58	Catalyst bank 1, manufacturer defined test

MID	Record	Meaning
A2, A3	0B 24 00 00 00 00 64	Misfire counts per cylinder of the boxer twin

Each record is test ID, unit and scaling ID, then value, minimum and maximum as 16-bit words; the runtime prefixes the monitor ID. Requests 06 00, 06 20 and 06 A0 return the support bitmaps that chain the ranges, generated from the declared IDs. The failing 02 record is what P0130 above reports.

Oxygen sensor tests and control operations stay empty, so mode 05 and mode 08 answer only their support bitmap and refuse concrete IDs.

8. Load and test

Studio

Open the copied vehicle wrapper, confirm BMS-X with two handlers, and enable the Internal Bus. Select and enable External CAN for a physical bench. Set the adapter's bit rate to match your tester.

Native CLI: direct bridge

Open a new Terminal window after installing Studio so `ecum` and `ecp` are on PATH.

Activate your pilot license in Studio first, or run `ecum --license-import /path/to/customer.license` as the same OS user. Check activation with `ecum --license-status`. A remote runtime needs a license on its own host; Studio does not transmit your license when connecting.

```
ecum --vehicle "$HOME/Documents/ECUmulator/motorrad/bmw-motorrad-vehicle.json" --socketcan gs_usb
```

The direct bridge uses 500 kbit/s Classic CAN and stops with Ctrl-C. The short selector chooses the first `gs_usb` adapter. Narrow it to **gs_usb:1d50:606f** or use a concrete inventory ID if several match. Other backend names include `toucan`, `openport`, `peak_usb_fd` and `socketcan` (Linux).

Native CLI: control server

In the first terminal:

```
ecum --control-host 127.0.0.1 --control-port 29190
```

In a second terminal:

```
ecp --host 127.0.0.1 --port 29190 load "$HOME/Documents/ECUmulator/motorrad/bmw-motorrad-vehicle.json"
ecp --host 127.0.0.1 --port 29190 can-interfaces
ecp --host 127.0.0.1 --port 29190 patch externalCan
'{"interface":"gs_usb","bitrate":500000,"canFd":false,"enabled":true}'
```

Direct bridge and control server are separate modes: do not combine `--socketcan` and `--control-port`. `ecp` controls the runtime; it is not a diagnostic CAN tester.

Expected responses

These are diagnostic payloads **without ISO-TP framing**. Send physical requests on 0x7E0 and receive on 0x7E8.

Request	Expected response	Check
01 05	41 05 82	Live coolant: 90 °C
01 0C	41 0C 1F 40	Engine speed: 2000 rpm

Request	Expected response	Check
01 0D	41 0D 32	Vehicle speed: 50 km/h
01 5C	41 5C 6E	Engine oil temperature: 70 °C
01 42	41 42 36 B0	Module voltage: 14 V
01 F1	41 F1 12 34	Custom lab integer: 4660
01 F2	41 F2 44 45 4D 4F	Custom lab text: DEMO
22 F1 00	62 F1 00 1F 40	UDS engine speed: 2000 rpm
02 0C 00	42 0C 00 0F A0	Freeze frame: 1000 rpm
22 F1 91	62 F1 91 48 57 30 31	Hardware version: HW01
22 F1 90	62 F1 90 + 17 ASCII bytes	UDS VIN
09 02	49 02 01 + 17 ASCII bytes	OBD-II VIN

Mode 02 responses echo the requested freeze-frame index after the PID, before the snapshot data:
42 PID FRAME DATA....

Without hardware, inject a complete Classic CAN single frame:

```
ecum --vehicle "$HOME/Documents/ECUmulator/motorrad/bmw-motorrad-vehicle.json" --request '7E0#0201050000000000'
```

Expect a reply containing **03 41 05 82**. The request's leading 02 is the ISO-TP length. Long responses need flow control, so use an ISO-TP tester for VIN checks.

9. Create a variant

1. Copy all three files.
2. Change the vehicle name and logical ECU names.
3. Adjust request and reply IDs together.
4. Update all VIN representations.
5. Change one static DID/PID and verify its exact response.
6. Add sequences or generators after the static version works.
7. Reload and repeat the checks after every edit.

Keep an untouched demo copy to distinguish transport faults from spec changes. Keep encoding formulas beside non-obvious values. A small verified scenario is a better starting point than many unverified fields.

10. Troubleshooting

Symptom	First checks
JSON fails to load	Quotes, commas, brackets; no JSON5; inspect the reported file
ECU file missing	Copy all files; paths are relative to the vehicle
Two separate ECUs	Vehicle entries must have exactly the same name
No diagnostic reply	Loaded handler, enabled ECU/bus, adapter, bit rate, request ID

Symptom	First checks
VIN stops after first frame	Tester must send ISO-TP flow control
Incorrect value	Raw bytes vs. physical units; scale and byte order
Unexpected response prefix	Do not include service/PID/DID or ISO-TP headers in data
Sequence skips values	Another client may be polling it
File edit has no effect	Reload/apply the edited vehicle; check the active target
No CAN frame console trace	Direct-bridge frame logs are Debug; normal output is Info

Printing and sharing

This guide is part of the PDF manual: choose **Open PDF manual** in the help browser or **Help** → **PDF Manual** to read or print it. The complete demo files follow in [Example Spec Files](#).

The ecum Runtime

ecum is the native runtime engine that emulates the ECUs of a vehicle specification. Studio starts and controls it for you; from a terminal it runs on its own, on a headless test machine, or as a control server that Studio or [ecp](#) connect to over the network. ecumulatorpp is the same program under its full name.

On macOS, the installer puts ecum on the PATH of new terminal windows and installs its man page: `man ecum` shows the complete reference. This chapter follows that man page.

License

Desktop emulation requires the same signed offline license as Studio. Both share the per-user `studio.license` file, so a license activated in Studio also activates ecum for the same OS user. On a machine without Studio:

```
ecum --license-import /path/to/customer.license
ecum --license-status
```

`--license-status` exits nonzero when no valid license is installed. A control server checks the license of its own host before changing anything; read-only status, stop and shutdown work without one. Expiry blocks new starts and changes but never interrupts a running session. `--version` prints the version without a license.

Operating modes

Exactly one mode must be selected:

Mode	Option	Purpose
Request injection	<code>--request ID#DATA</code>	Send fixed tester frames and print the responses, then exit
Direct CAN bridge	<code>--socketcan SELECTOR</code>	Connect the virtual bus to a physical CAN adapter

Mode	Option	Purpose
ELM327/STN adapter	<code>--elm327-tcp PORT</code>	Serve an interactive ELM327/STN2xxx-compatible TCP adapter
ECUconnect adapter	<code>--ecuconnect-tcp PORT</code>	Serve the ECUconnect TCP adapter protocol
Control server	<code>--control-port PORT</code>	Serve the JSON control API for Studio and ecp

All modes except the control server need `--vehicle PATH` with a vehicle or ECU JSON file. The control server can start empty and receive its vehicle later. Server modes run until interrupted or until `--duration-ms` expires.

Options

Option	Meaning
<code>--vehicle PATH</code>	Vehicle preset JSON. Required unless <code>--control-port</code> is used
<code>--request ID#DATA</code>	Inject one tester CAN frame, e.g. 7DF#0201000000000000. Repeat it for multi-frame exchanges
<code>--socketcan SELECTOR</code>	Bridge the vehicle to a CAN adapter at 500 kbit/s Classic CAN
<code>--duration-ms MS</code>	Stop server modes after this many milliseconds. Default: run until interrupted
<code>--elm327-tcp PORT</code>	Run the ELM327/STN2xxx-compatible TCP service on this port
<code>--elm327-host ADDRESS</code>	Bind address of the ELM327 service. Default: 127.0.0.1. <code>--host</code> is an alias
<code>--ecuconnect</code>	Enable the ECUconnect TCP adapter on its default port 8129
<code>--ecuconnect-tcp PORT</code>	Enable the ECUconnect TCP adapter on an explicit port
<code>--ecuconnect-host ADDRESS</code>	Bind address of the ECUconnect adapter. Default: 0.0.0.0
<code>--ecuconnect-serial SERIAL</code>	Adapter serial number reported by the ECUconnect service
<code>--ecuconnect-logger</code>	Stream bus frames to CANcorder and ECUconnect logger clients on port 2518
<code>--ecuconnect-logger-tcp PORT</code>	Logger stream on an explicit port
<code>--ecuconnect-logger-host ADDRESS</code>	Bind address of the logger stream. Default: 0.0.0.0
<code>--ecuconnect-logger-name NAME</code>	Zeroconf instance name of the logger service
<code>--no-ecuconnect-logger-zeroconf</code>	Do not publish the logger stream via Zeroconf
<code>--can-log FILE or DIRECTORY</code>	Append every bus frame to a candump trace; see below

Option	Meaning
<code>--can-log-channel NAME</code>	Interface column of the trace, written as NAMErx and NAMEtx. Default: ecum
<code>--can-log-capacity FRAMES</code>	In-memory queue depth, at least 1024. Default: 65536 (about 5 MB)
<code>--control-port PORT</code>	Run the JSON control API on this TCP port
<code>--control-host ADDRESS</code>	Bind address of the control API. Default: 127.0.0.1
<code>--control-name NAME</code>	Zeroconf instance name of the control service
<code>--control-zeroconf, --no-control-zeroconf</code>	Publish the control service via Zeroconf (default) or not
<code>--owner-pid PID</code>	Exit when this process disappears; Studio uses it for its managed runtime
<code>--license-import FILE</code>	Validate and install a license file for the current user
<code>--license-status</code>	Show the activation status
<code>--version, -h, --help</code>	Print the version or the usage and exit

CAN adapters

`--socketcan` and the control API's External CAN interface accept the same selectors. A backend name selects the first adapter of that kind:

Selector	Adapter
<code>gs_usb</code>	CANable/candleLight-compatible adapters (macOS)
<code>toucan</code>	Rusoku TouCAN (macOS)
<code>openport</code>	Tactrix OpenPort, e.g. <code>openport:/dev/cu.usbmodem...</code> (macOS)
<code>peak_usb_fd</code>	PEAK USB FD adapters (macOS)
<code>socketcan, can0, vcan0</code>	SocketCAN interfaces (Linux)
<code>ecuconnect:HOST[:PORT]</code>	ECUconnect adapter over Wi-Fi or USB-Ethernet, port 129 by default

Complete leading fields narrow the choice, e.g. `gs_usb:1d50:606f`; a concrete inventory ID selects one specific adapter. Fields match as a whole: address 1 does not match 12.

An ECUconnect **adapter** on port 129 is a CAN interface. Do not confuse it with port 8129, where the emulator's own ECUconnect **service** offers the same protocol to a diagnostic app. The adapter serves one connection at a time over TCP; Bluetooth LE cannot carry the request rate of a vehicle scan.

After a connection failure, the bridge retries the original adapter with backoff of up to five seconds and restores its CAN configuration. USB adapters are recognized by serial number, or by physical port without one. Frames sent while disconnected are not replayed. A failure while first opening the adapter still ends the program immediately.

Recording a CAN trace

`--can-log` appends every frame the virtual bus accepts, in both directions and including traffic no emulated ECU handles, to a `candump -L` trace. It starts with the process, needs no connected client and survives a crash of the emulator. Frames are queued without blocking and written by a low-priority thread; if the queue overflows, whole frames are dropped and the count is reported at exit.

Given an existing directory (. for the current one), the file is named `ecum-SESSION-YYYYMMDD-HHMMSS.log`, where `SESSION` is the vehicle name, the CAN adapter or `control`. A generated name is never reused. In `control-server` mode, loading another vehicle starts a new file named after it, and clients can add marker comments. Any other argument is used as a file name and appended to.

The logger stream (`--ecuconnect-logger`) is the live view for `CANcorder` and only records while a client is attached. Console output shows startup at Info level; individual frames are logged at Debug level, which requires a build configured with a higher console log level.

Examples

Send one OBD-II request and print the response:

```
ecum --vehicle /usr/local/cansole/share/specs/can/obd2-can11-vehicle.json \  
--request '7DF#02011C0000000000'
```

Read the VIN via UDS ReadDataByIdentifier, including the ISO-TP flow-control frame that the multi-frame response needs:

```
ecum --vehicle /usr/local/cansole/share/specs/default-vehicle.json \  
--request '7E0#0322F19000000000' \  
--request '7E0#3000000000000000'
```

Serve a vehicle as an ELM327/STN adapter on port 35000:

```
ecum --vehicle /usr/local/cansole/share/specs/can/obd2-can11-vehicle.json \  
--elm327-tcp 35000
```

Bridge a vehicle to the first `gs_usb` adapter and record a trace in an existing directory:

```
ecum --vehicle /usr/local/cansole/share/specs/default-vehicle.json \  
--socketcan gs_usb --can-log ~/Documents/CANsole/Traces
```

Run a control server that Studio or `ecp` can use from another computer:

```
ecum --control-host 0.0.0.0 --control-port 29190
```

A control server binds to 127.0.0.1 by default and then accepts local clients only. The direct bridge and the control server are separate modes: do not combine `--socketcan` with `--control-port`; enable External CAN through the control API instead, as shown in [The `ecp` Remote Control](#).

Other installed tools

The macOS installer also installs these tools with their man pages. Studio uses the importers behind **File** → **New....**

Tool	Purpose
<code>canlog-importer</code>	Generate a vehicle/ECU spec bundle from CAN log captures
<code>odx-importer</code>	List the variants of an ODX/PDX package or generate an ECU spec for one

Tool	Purpose
toucan-cli	List Rusoku TouCAN adapters, dump and send classic CAN frames
openport-cli	Check Tactrix OpenPort adapters on CAN and K-Line

The ecp Remote Control

ecp drives a running [ecum](#) control server from a shell or a script: load a vehicle, switch services, arm or disarm ECUs, inspect CAN interfaces, read a physical vehicle, or watch everything live in a terminal user interface. It uses the same control API as Studio, so both can work with the same runtime at the same time.

ecp is a native program; it needs no Python, Homebrew or developer tools. `man ecp` shows the complete reference, which this chapter follows. ecp relies on the license of the runtime it controls.

Finding the runtime

Without endpoint options, ecp discovers compatible runtimes through DNS-SD and asks which one to use when there are several. With `--host` or `--port`, the missing value defaults to 127.0.0.1 or 29190. The environment variables `ECUM_CONTROL_HOST` and `ECUM_CONTROL_PORT` work the same way.

Run without arguments on an interactive terminal, ecp opens the terminal user interface of the discovered runtime.

Global options

Option	Meaning
<code>--host HOST</code>	Control API host. Default: 127.0.0.1
<code>--port PORT</code>	Control API port. Default: 29190
<code>--timeout SECONDS</code>	Request timeout. Default: 3.0
<code>--json</code>	Print raw JSON instead of a formatted table, where supported
<code>--no-input</code>	Never prompt; several discovered runtimes then need an explicit host and port
<code>--no-color</code>	Disable terminal colors; <code>NO_COLOR</code> is honored too
<code>--version</code>	Print the client version without connecting
<code>-h, --help</code>	Print the usage and exit

Commands

Command	Purpose
<code>help [COMMAND]</code>	General help or help for one command
<code>list</code>	Configured service states
<code>ecus</code>	Runtime arming state of each ECU
<code>get</code>	Current service configuration
<code>config</code>	Complete current runtime configuration
<code>capabilities</code>	Service capabilities the runtime advertises
<code>health</code>	Health counters of the runtime
<code>can-interfaces</code>	CAN interfaces the runtime detects
<code>catalog [--vehicle-dir DIR]</code>	Loadable vehicle and ECU specs in a directory
<code>service-options</code>	Metadata for service option pickers
<code>load FILE [--reset-options]</code>	Load a vehicle and its referenced ECU files
<code>enable SERVICE..., disable SERVICE...</code>	Switch one or more services, e.g. hsfz, doip; --dry-run shows the change only
<code>enable-ecu ECU, disable-ecu ECU</code>	Arm or disarm one loaded ECU; --protocol and --request-id pick one handler
<code>patch SERVICE JSON</code>	Merge-patch the options of one service; --dry-run shows the change only
<code>analyze --interface ADAPTER</code>	Read a physical vehicle into a spec; see below
<code>analysis status, cancel, result</code>	Inspect, stop or recover the latest analysis
<code>tui</code>	Interactive service and ECU monitor
<code>discover</code>	Compatible runtimes found via DNS-SD
<code>rpc METHOD [--params JSON]</code>	Call an advertised control API method directly, for diagnostics

`load` resolves the ECU files a vehicle refers to and uploads their contents, so the runtime does not need access to your paths. Existing service and bus options stay as they are unless `--reset-options` is given. A single ECU file is accepted too and wrapped into a one-ECU vehicle.

Service names are those of the control API: `canBus` (Internal Bus), `externalCan`, `hsfz`, `doip`, `elm327`, `ecuconnect`, `enetServiceBroker`, `logger` and `externalKLine`. `ecp list` shows the ones a runtime offers. For `externalCan`, the interface accepts the adapter selectors described in [The ecum Runtime](#), such as `gs_usb` or `gs_usb:1d50:606f`.

Examples

Show the services of a local runtime, load a demo vehicle and enable HSFZ:

```
ecp --port 29190 list
ecp --port 29190 load /usr/local/cansole/share/specs/motorrad/bmw-motorrad-vehicle.json
ecp --port 29190 enable hsfz
ecp --port 29190 disable hsfz doip
```

Change the HSFZ port and bridge the runtime to a physical CAN bus:

```
ecp --port 29190 patch hsfz '{"tcpPort":6801}'
ecp --port 29190 patch externalCan '{"interface":"gs_usb","bitrate":500000,"canFd":false,"enabled":true}'
```

Open the terminal user interface with the installed demo catalog:

```
ecp --port 29190 tui --vehicle-dir /usr/local/cansole/share/specs
```

Checking physical CAN health

`ecp list` and `ecp tui` show the External CAN controller state and supported TEC/REC, error-report, ACK, protocol, bus-off and queue-loss counters. The adapter's connection state is separate: an active connection can report CAN=bus-off. `ecp --json list` preserves the complete `bus_health` object, including unavailable counters as `null`. A stale sample is labelled explicitly. An open adapter with zero received frames is marked CAN RX unconfirmed; error-active alone cannot establish physical bus contact. `ecp health` reports runtime health counters; physical CAN health belongs to the External CAN row. See [External CAN](#) for interpretation and adapter limitations.

Reading a vehicle

`ecp analyze` runs the same analysis as Studio's [Read from Vehicle](#) over CAN, DoIP or HSFZ and writes an inline vehicle spec:

```
ecp --host localhost analyze --interface can0 --name "My vehicle" \
  -o vehicle.json --report report.json
ecp --host localhost analyze --interface gs_usb --deep \
  --route 0x700:0x708 -o vehicle.json
ecp --host localhost analyze --transport doip -o vehicle.json
ecp --host localhost analyze --transport hsfz --ethernet-interface en5 -o vehicle.json
ecp --host localhost analysis status
```

Option	Meaning
<code>--transport can doip hsfz</code>	Connection type; CAN by default
<code>--interface ADAPTER</code>	External CAN adapter of the runtime; required for CAN
<code>--ethernet-interface NAME</code>	Search one Ethernet interface on the runtime host; omitted searches all, with link-local preferred
<code>--vehicle-host HOST, --vehicle-port PORT</code>	Optional manual gateway bypasses UDP discovery; ports default to 13400 for DoIP, 6801 for HSFZ
<code>--tester-address ADDRESS</code>	Tester identity; DoIP 0xE80, HSFZ 0xF4, CAN passive discovery 0xF9
<code>--functional-address ADDRESS</code>	Functional diagnostic address; DoIP 0xE400, HSFZ 0xDF; manufacturer-specific
<code>--target ADDRESS</code>	Repeatable physical ECU address for DoIP/HSFZ; replaces broadcast discovery

Option	Meaning
<code>--activation-type TYPE</code>	DoIP routing activation type; 0 by default
<code>--discovery-timeout-ms MS</code>	Collection window for all functional replies; 1500 ms by default
<code>--max-parallel-ecus N</code>	How many ECUs are asked at once, one request each; 8 by default over DoIP/HSFZ, 1 over CAN (at most 16 there)
<code>--functional-sweep</code>	DoIP/HSFZ, experimental: read the F1xx identifiers with one functional request each instead of per ECU
<code>-o FILE, --report FILE</code>	Write the vehicle spec and the report. Without <code>-o</code> , the spec goes to standard output
<code>--depth LEVEL, --deep</code>	basic (discovery only), standard (default) or deep, which tries every identifier and may take hours
<code>--route REQUEST:REPLY</code>	Repeatable address pair that replaces automatic discovery
<code>--name NAME</code>	Vehicle name in the spec
<code>--bitrate, --can-fd, --data-bitrate</code>	CAN bus settings
<code>--timeout-ms, --interval-ms</code>	Response timeout and pause between requests; no pause by default
<code>-q, -f</code>	Suppress progress; overwrite existing files

The analysis uses default sessions and read requests only; OBD-II modes 04 (clear faults) and 08 (actuator control) are never sent. For CAN scans, the simulation is disconnected from External CAN and stays disconnected afterwards. Progress goes to standard error; `--json` returns the full result envelope on standard output. Ctrl-C cancels and saves the partial result.

For DoIP/HSFZ, UDP discovery first locates the vehicle gateway. A single result is selected automatically. Multiple results are listed with VIN, IP and interface; select one using `--vehicle-host` and `--ethernet-interface`. The TCP connection uses the selected interface. `ecp rpc list_ethernet_interfaces` lists eligible interfaces on the runtime host.

Every ECU answering functional diagnostic requests is read individually. An early response never closes the collection window. Functional addresses and gateway access depend on the manufacturer; silent or protected ECUs may not be found. Use `--target` for known addresses. TCP acknowledgements are excluded. DoIP routing activation must succeed; TLS and authentication are not implemented. The connection closes after the scan and External CAN settings are unaffected. `--host` selects the runtime; `--vehicle-host` selects the car.

Exit status	Meaning
0	Analysis complete
1	Analysis failed; a partial spec was saved
2	No identifiable ECU answered
130	Cancelled with Ctrl-C; partial results saved

Results stay in the runtime's memory until the next analysis or until it exits. After a disconnected client, `ecp analysis result` recovers them and `ecp analysis cancel` stops a scan that is still running.

The terminal user interface

`ecp tui` lists services and ECUs side by side. Tab switches between the two tables, the arrow keys or j/k select a row, Space or Enter toggles it. `v` opens the searchable vehicle catalog, and enabling External CAN opens the interface picker. `r` resets the counters, `R` reconnects and `q` quits. Controls are disabled while disconnected, and a disconnected snapshot is marked as stale.

Active services are shown in green, idle ones in yellow, errors in red and unavailable services dimmed; the state labels also work without color. `--health-interval SECONDS` sets the refresh rate, and `--log-file FILE` (or `ECP_TUI_LOG_FILE`) shows the tail of the runtime's console log at the bottom. When the runtime records a CAN trace, the interface shows its path.

Configuration file

`~/.config/ecp/config.toml` can set the default vehicle directory:

```
vehicle_dir = "~/specs"
```

`ECP_CONFIG` selects another file, `XDG_CONFIG_HOME` another base directory. The vehicle catalog directory is taken from `--vehicle-dir`, then `ECP_VEHICLE_DIR`, then the configuration file, and finally the installed catalog in `/usr/local/cansole/share/specs`.

Vehicle Scan Reference

Use this chapter when the defaults in [Read from Vehicle](#) do not fit your vehicle, or when you need the exact captured ranges. Start with the task guide for connection, inspection depth and saving results.

Ethernet discovery and routing

Choose **Ethernet interface** to search one active interface, or **Automatic** to broadcast over all active Ethernet interfaces. IPv4 link-local addresses are tried first and preferred when the same gateway answers through multiple links. The interfaces belong to the connected runtime host, including remote runtimes. UDP discovery uses port 13400 for DoIP and 6811 for HSFZ.

One discovered vehicle is selected automatically, including when the same gateway answers through several interfaces. If several distinct vehicles answer, select one by VIN, address and interface. The list updates automatically, with a three-second pause between searches. Connecting Ethernet or switching on the ignition needs no extra click. Searches stop when you start the analysis, close the dialog, select CAN or enter a gateway manually. The diagnostic TCP connection uses the same interface and local address as the selected discovery result. **Enter gateway address manually** allows a host and TCP port for gateways that cannot be discovered. No External CAN adapter is required.

Setting	DoIP	HSFZ
TCP port	13400	6801
Tester address	0x0E80	0xF4

Setting	DoIP	HSFZ
Functional discovery address	0xE400	0xDF
Routing activation type	0	Not applicable

Under **Physical addresses · Timing**, these values can be changed for the vehicle. DoIP functional group addresses are manufacturer-specific; 0xE400 is the initial setting, not a universal inventory address. Successful routing activation is required before DoIP diagnostic requests are sent. Gateways that require authentication, TLS or another access procedure need that support before they can be read by this client.

Discovery sends functional Tester Present (3E 00 and KWP 3E), VIN read (22 F1 90), BMW ECU software number (22 F1 01), KWP identification (1A 80) and OBD2 support (01 00). It collects all matching ECU responses for **1500 ms per request** by default, independently of adaptive timing. The first response never ends the collection window. Every physical responder is then inspected separately with the selected depth. Transport acknowledgements are not ECU responses, but once a gateway sends them, each window starts with its request's acknowledgement: a request that waited in the gateway does not lose that time. Response-pending (7F ... 78) extends the wait for that ECU only, until it answers or its P2* has passed.

Over DoIP and HSFZ the gateway forwards an answer only once it has all of it, so a long answer (a large fault memory, for example) arrives well after the ECU started sending it. A physical request to an ECU that has already answered therefore waits up to that ECU's own P2* (from its 10 01 answer, 5 s by default) before it counts as unanswered. This costs nothing while the ECU answers. The next request goes out as soon as an answer arrives; a retry after a missed answer goes out at once.

Over DoIP and HSFZ the scan asks several ECUs at once, one request each: **ECUs at once** under Physical addresses · Timing, 8 by default. A functional request is always sent alone. When an ECU answers busy (7F xx 21) or the gateway reports it is out of memory, the scan asks fewer ECUs at once, down to one; an ECU the gateway reports unreachable is skipped and named in the warnings. Set **ECUs at once** to 1 to compare with a one-at-a-time scan.

Over CAN, **ECUs at once** starts at 1. Higher values (up to 16) are faster with USB adapters such as gs_usb; ECUconnect sends on one CAN id at a time and pays a round trip over the network for every change, so it gains less. While several ECUs answer at once, the scan asks them for a 1 ms pause between the frames of long answers to keep the bus load down.

Read identifiers by broadcast (experimental) reads the F100–F1FF block with one functional request per identifier instead of one per ECU. An ECU without the identifier stays silent on a functional request, so every identifier waits for the slowest ECU's window. It reaches only ECUs the gateway routes functional requests to. Use it to compare time and result with a normal scan on the same vehicle.

BMW F101 responses with a valid SVK layout are decoded into hardware, bootloader, software and coding version identifiers. These appear in the ECU identity while the scan runs and in the exported identifier's comment. The original response bytes are preserved for emulation.

To read known ECUs directly, enter their addresses under **Physical ECU addresses**, separated by commas. This replaces broadcast discovery. Decimal and 0x-prefixed hexadecimal values are accepted. HSFZ uses 8-bit addresses; DoIP uses 16-bit logical addresses. The ECU has the same logical address in requests and responses; exported specs preserve it and the functional address.

A broadcast finds only ECUs that answer and are routed by the gateway. Silent, protected or sleeping ECUs may remain undiscovered. UDP vehicle announcements identify gateways. After selecting a vehicle, functional diagnostic requests find the ECUs behind its gateway.

The dedicated connection closes on completion, cancellation or failure. Partial results remain exportable. The simulator's External CAN settings are unaffected by a DoIP/HSFZ scan. Configuration changes remain blocked while analysis runs.

Requests by inspection depth

Depth	Requests
Standard	Discovery (default diagnostic sessions and Tester Present, KWP ECU identification, UDS VIN, OBD2 01 00), then the KWP identification options the ECU lists in its scaling table 1A 81 (80-9F without one) and fault memory, UDS identification F100-F1FF and fault memory, OBD data over UDS (F4xx/F6xx/F8xx, following their bitmaps), supported OBD2 data and DTCs
Deep	Standard, every 22 0000-FFFF identifier, plus KWP 1A 00-FF and 21 00-FF, and the complete UDS DTC list (19 02 FF)

For BMW group (BMW, Mini) and VW group (VW, Audi, Seat, Skoda, Porsche) vehicles, recognised by the first three characters of the VIN, Standard also reads the manufacturer's identifiers outside F1xx from every UDS ECU: 17 for BMW (energy mode 100A, programming counters 2502/2503, SVK reference 8008 ...) and 21 for VW (coding 0600, programming counters 0407-040A, flash state 0405, mileage 295A, SFD status 0174 ...). Exported identifiers carry the manufacturer's names (F102 SVK *System Supplier*, F187 VW *Spare Part Number*), and BMW's further software lists (F102, F103, F141, F142) are decoded like F101. KWP ECUs are not asked; they number their identifiers differently.

Standard reads the UDS fault memory with status mask 0C: pending and confirmed DTCs, as BMW testers read it. It leaves out codes that have not failed; with FF ECUs list their whole DTC catalogue, hundreds of entries on some BMW ECUs. When an ECU answers that the list is too long for one response (7F 19 14), the scan reads confirmed DTCs (08) instead, then pending ones (04), and marks the ECU's fault memory as partial. KWP fault memory is read as stored (18 02 FF FF) and pending (18 11 FF FF) DTCs. An ECU whose fault memory could not be read is named in the scan's warnings: the exported spec cannot tell "not read" from "empty", and its replica reports no faults.

Studio offers Standard and Deep. The command-line client also offers --depth basic for discovery alone.

Standard reads OBD2 modes **01, 02, 03, 05, 06, 07, 09 and 0A**. PID bitmaps determine which data is requested, including subsequent bitmap pages. Mode 02 captures frame 0 and then consecutive additional freeze frames until the first missing or empty frame, up to frame 255. Export preserves their frame indexes. Mode 05 is attempted but may be unsupported on CAN. Mode **04 (clear faults)** and mode **08 (actuator control)** are excluded. Analysis never sends security access, writes, routines, resets or DTC clearing.

CAN addresses and timing

On CAN, automatic discovery first listens to the bus for two seconds without sending anything. On a J1939 bus, such as a truck's, every ECU announces its source address, and its diagnostics live at 18DA<address>F9; those addresses are probed. Then come the OBD2 functional requests on 7DF and 18DB33F1, and, unless a 29-bit ECU has answered by then, the physical request IDs 7E0-7E7 with response IDs 7E8-7EF. Every responder is inspected individually. It does not scan the entire CAN address space or detect the bitrate automatically; an ECU that neither announces itself nor answers the OBD2 broadcast needs a custom address pair.

For manufacturer-specific diagnostic addresses, expand **Physical addresses · Timing**, select custom pairs and enter one request:reply pair per line:

```
0x700:0x708  
0x18DA42F1:0x18DAF142
```

Custom pairs replace automatic discovery. Both IDs must have the same width and reply IDs must be unique. CAN supports normal ISO-TP addressing on Classic CAN and CAN FD. TP2.0, K-line and ISO-TP extended/mixed address bytes are not supported.

The default response timeout is 250 ms without adaptive timing. With it (the default) each ECU's window follows its own P2 and measured answers. There is no pause between an answer and the next request unless one is configured. Each response-pending reply restarts the ECU's P2* wait, up to 30 seconds per request. Increase the timeout for a slower vehicle; negative responses and timeouts appear in the report, along with a log of every request and its outcome.

Export fidelity

The spec preserves observed identifier bytes, supported OBD2 data and stored/pending/permanent OBD2 faults. UDS fault reads also request 19 01 FF and retain the reported DTC format in the spec's `dtcFormatIdentifier` field. Specs without this field retain the existing ISO 14229 format default (01). It is a snapshot: it cannot reconstruct dynamic signals, security algorithms, inaccessible identifiers or behavior that was not observed. A positive 22 response alone cannot distinguish KWP from UDS; such classifications are marked **inferred**. Review these and the native interpreter defaults before using the spec as a vehicle model.

For the command-line workflow, options, progress and exit codes, see [The ecp Remote Control](#). To continue with an exported model, return to [Connect Your Tester](#).

Vehicle Files

Vehicle project files define a complete vehicle configuration with multiple ECUs.

For ECUmulator++, Studio compiles ECU files, legacy include-style vehicle files, and inline vehicle configs into a resolved runtime bundle before upload/apply. The authoring formats stay convenient; the runtime format is canonical.

Overview

A vehicle project references one or more ECU spec files and includes vehicle-level metadata.

Vehicle-level bus settings such as transport profile also live here.

Basic Structure

```
{
  "name": "BMW E90 330i",
  "vin": "WBADE6324VBW12345",
  "bus": {
    "mode": "virtual",
    "bitrate": 500000,
    "timingFactor": 0,
    "transportProfile": "isotp"
  },
  "ecus": [
    {
      "name": "DME",
      "path": "./dme-ecu.json"
    },
    {
      "name": "EGS",
      "path": "./egs-ecu.json"
    }
  ]
}
```

Top-Level Fields

Field	Type	Description
name	string	Vehicle display name
vin	string	17-character VIN (optional)
info	string	Optional author-written description of the purpose, emulated functions and limitations; plain text or Markdown
bus	object	Vehicle-level CAN/transport settings
ecus	array	List of ECU references
kline	object	Optional logical K-Line binding and addressed ECUs

Use the **Info** button beside **Vehicle** to open the Markdown editor. Choose source, preview or both side by side. **Apply** updates the vehicle; **Cancel** discards the draft. Save the vehicle file to keep the changes. The text is part of the vehicle specification and survives saving, runtime upload, reconnect and sharing. All included demo vehicles describe their intended use and scope here. Existing specifications without info remain valid. ECU source files can also provide an info string; Studio uses it when opening the ECU as a standalone vehicle.

Bus Settings

Typical bus block:

```
{
  "bus": {
    "mode": "virtual",
    "bitrate": 500000,
    "timingFactor": 0,
    "transportProfile": "sae_tp20",
    "tp20": {
      "testerId": "0x200",
      "applicationType": "0x01"
    }
  }
}
```

Supported transportProfile values:

- isotp
- isotp_fd
- sae_tp20

For TP2.0 vehicles, ECU specs also need ECU-level transport.tp20 endpoint metadata.

ECU References

Each ECU entry points to an external spec file:

```
{
  "name": "Engine Control Module",
  "path": "./engine-ecu.json"
}
```

- **name** — Display name / role in the UI
- **path** — Relative path to ECU spec file

Path Resolution

Paths are resolved relative to the vehicle project file:

```
vehicle/
├─ my-vehicle.vehicle
├─ engine-ecu.json
└─ transmission-ecu.json
```

In my-vehicle.vehicle:

```
{
  "ecus": [
    { "name": "Engine", "path": "./engine-ecu.json" },
    { "name": "Transmission", "path": "./transmission-ecu.json" }
  ]
}
```

VIN Format

The VIN should be a valid 17-character Vehicle Identification Number:

- Characters 1–3: World Manufacturer Identifier (WMI)
- Characters 4–9: Vehicle Descriptor Section (VDS)

- Characters 10–17: Vehicle Identifier Section (VIS)

Invalid characters (I, O, Q) are automatically rejected.

Loading

When loading a vehicle project:

1. ECUmulator reads the project file
2. Each ECU path is resolved and loaded
3. ECUs are added to the virtual CAN bus
4. Services begin listening for requests

Saving

When saving from the UI:

- Single ECU configurations save as ECU spec files
- Multi-ECU configurations save as vehicle project files

Sample Files

ECUmulator includes sample vehicle project files in the `specs/` directory:

- `default-vehicle.json` — Demo vehicle configuration (legacy sample)
- `obd2-can29-vehicle.json` — OBD-II with extended addressing (legacy sample)
- `multi-protocol-ecu-vehicle.json` — one ECU answering both UDS and OBD-II on the same arbitration ID (two `ecus[]` entries sharing a name)

Inline ECU Definitions

For simple cases, ECU configuration can be inline:

```
{
  "name": "Simple Vehicle",
  "vin": "WBADE6324VBW12345",
  "ecus": [
    {
      "name": "ECU",
      "kind": "uds",
      "uds": {
        "arbitrationInfo": {
          "request": "0x7E0",
          "reply": "0x7E8"
        },
        "identifiers": [...]
      }
    }
  ]
}
```

However, external files are recommended for maintainability.

Studio Note

Studio preserves `bus.tp20` and `ECU transport.tp20` blocks when loading and applying vehicle files. It does not yet provide a dedicated TP2.0 editor, so those fields are currently best maintained in the JSON file itself.

For ECUmulator++ transport and remote control, Studio compiles project files and ECU specs into a canonical runtime bundle envelope before upload/apply. The saved file and the runtime payload are intentionally different artifacts.

K-Line binding

A vehicle can also bind existing OBD-II/KWP ECUs to one physical K-Line:

```
"kline": {
  "bus": "diagnostic",
  "ecus": [
    { "ecu": "Engine", "address": 16 },
    { "ecu": "Transmission", "address": 24 }
  ],
  "initMode": "auto",
  "initAddress": 51,
  "testerAddress": 241,
  "baud": 10400,
  "keywords": [233, 143],
  "responseDelayUs": 25000
}
```

Names refer to vehicle ECUs. Up to 16 distinct ECU addresses are supported; none may equal the tester or functional initialization address. `initMode` is `auto`, `fast` or `slow`; response delay is 25000–49000 microseconds. The current profile fixes baud and keywords to the values above. Legacy `ecu/ecuAddress` bindings remain readable; do not combine them with `ecus`.

The binding is saved with the vehicle. Enabling the hardware is a separate runtime service setting (`externalKLine`, `interface onboard`, `matching bus`). See [K-Line on ECUconnect](#) for remote setup and demo checks.

ECU Spec Files

ECU spec files define individual ECU configurations in JSON format.

Overview

An ECU spec file contains:

- Protocol type (UDS, KWP, OBD-II)
- CAN arbitration information
- Protocol-specific configuration
- Optional custom service definitions for UDS/KWP

Basic Structure

```
{
  "kind": "uds",
  "name": "Engine Control Module",
  "uds": {
    "arbitrationInfo": {
      "request": "0x7E0",
      "reply": "0x7E8"
    },
    "identifiers": [...],
    "routines": [...],
    "securityLevels": [...],
    "customServices": [...],
    "dtcs": [...]
  }
}
```

Top-Level Fields

Field	Type	Description
kind	string	Protocol: uds, kwp, or obd2
name	string	ECU display name
uds/kwp/obd2	object	Protocol-specific configuration

Arbitration Info

Defines CAN message IDs:

```
{
  "arbitrationInfo": {
    "request": "0x7E0",
    "reply": "0x7E8",
    "auxiliary": ["0x7E1", "0x7E2"]
  }
}
```

- **request** — ID the ECU listens on
- **reply** — ID the ECU responds on
- **auxiliary** — Additional IDs (functional addressing)

UDS Configuration

```
{
  "uds": {
    "p2ServerMax": 0.05,
    "p2extServerMax": 2.0,
    "s3SessionTimeout": 5.0,
    "identifiers": [...],
    "routines": [...],
    "securityLevels": [...],
    "customServices": [...],
    "memoryLayout": [...],
    "dtcs": [...],
    "startupActions": [...]
  }
}
```

In Studio, the ECU Detail **Global Actions** UI writes firewall-related startupActions (lockAllOtherECUs / unlockAllOtherECUs) and is only shown when ECU role resolves to FIREWALL (case/separator-insensitive).

KWP Configuration

```
{
  "kwp": {
    "commonIdentifiers": [...],
    "ecuIdentifiers": [...],
    "localIdentifiers": [...],
    "routines": [...],
    "securityLevels": [...],
    "customServices": [...],
    "dtcs": [...]
  }
}
```

OBD-II Configuration

```
{
  "obd2": {
    "frames": [
      { "pids": [...] }
    ],
    "informational": [...],
    "oxygenSensorMonitoring": [...],
    "onBoardMonitoring": [...],
    "controlOperations": [...],
    "dtcs": [...]
  }
}
```

Custom Service Format (UDS/KWP)

Custom services let you define additional request handlers that run before built-in service handling.

```
{
  "customServices": [
    {
      "name": "Select Block",
      "matchType": "prefix",
      "request": "B100"
    },
    {
      "name": "ReadFastArray",
      "matchType": "strict",
      "request": "3E 33 50 03 B0 7C",
      "replay": [
        { "count": 1, "response": "7E BA 09 54 16 20 20" },
        { "count": 3, "response": "7E B7 09 54 16 20 20" }
      ],
      "replayLoop": true
    }
  ]
}
```

Custom Service Fields

Field	Type	Description
name	string	Display label
matchType	string	strict, prefix, or suffix
request	bytes	Request bytes to match
response	bytes	Fixed response payload
replay	array	Sequence responses with count and response
replayLoop	bool	Restart replay sequence when exhausted
delay	number	Delay in milliseconds before response
actions	array	Optional actions (same shape as security actions)
debugSwallowResponseLikeliness	number	Response drop probability 0.0 to 1.0

Notes:

- If both replay and response are present, replay is used first.
- If neither is present, a default positive response is generated.
- Custom-service sequencing is represented by replay; there is no separate sequence field.

Identifier Format

```
{
  "id": "0xF190",
  "name": "VIN",
  "type": "ascii",
  "content": "WBADE6324VBW12345"
}
```

Or with hex bytes:

```
{
  "id": "0xF188",
  "name": "Software Version",
  "type": "hex",
  "bytes": "01 02 03 04"
}
```

Security Level Format

```
{
  "level": "0x01",
  "delay": 10,
  "seedKeyPairs": [
    { "seed": "AA BB CC DD", "key": "11 22 33 44", "keyResponse": "34" }
  ]
}
```

DTC Format

```
{
  "code": "P0123",
  "status": "0x29",
  "description": "Throttle Position Sensor A Circuit High"
}
```

File Extension

ECU spec files typically use .json extension. The kind field determines the protocol.

Demo Vehicles

Choose a demo that matches your tester or the behaviour you want to model. For the first guided test, use [Your First Emulation](#). Loading a demo creates an editable copy; save your own version with **File** → **Save As....** The bundled originals stay unchanged.

Choosing an example

The top-level **Demo Vehicles** menu groups the catalog under **CAN**, **DoIP**, **HSFZ** and **TP2.0**, with separators between the groups. Categories and the vehicles within each category are sorted alphabetically. Handler badges (UDS, KWP, OBD-II) are derived from the actual ECU blocks once at startup; CAN FD examples also carry a CAN FD label. On macOS these use the native badge column; other platforms show bracketed labels.

Example	Purpose
BMW Motorrad — 1250 Boxer	Shared 11-bit ECU with two handlers; the first-emulation and authoring tutorials
Petrol Sedan — Engine & Automatic	Standalone OBD-II, 11-bit CAN, two ECUs
Diesel Van — Engine & Automatic	Standalone OBD-II, 29-bit CAN, two ECUs
Compact Car — Engine Identification	Small UDS identification baseline
Performance Hatchback — 2.0 Turbo	Two UDS ECUs over CAN FD; long records, guarded programming session, timed routines and a small memory transfer
Scania R-series — Diesel Tractor Unit	UDS engine and KWP gearbox on 29-bit CAN; shared VIN and timed self-tests
Porsche 911 — PDK Transmission	KWP over ISO-TP; advanced session/flash example
Volkswagen — V6 TDI	KWP over TP2.0, with explicit transport configuration
BMW 535d (2014) — 3.0 Diesel	Three UDS ECUs over HSFZ: ZGW, body domain (FEM_BODY), diesel powertrain
Audi Q7 (2020) — 3.0 V6 TDI	UDS over DoIP with logical address 0x4076

These vehicles cover the editable diagnostic features without crowding the menu with repeated controller variants. The hatchback and truck are synthetic composites: their info popups explain

their inspiration, addresses, test sequences and limits. Private captures and duplicate regression fixtures stay outside the customer catalog. Adapter selection, network discovery and physical timing still need their own transport or hardware checks.

Ethernet diagnostic demos

Load the matching vehicle from **Demo Vehicles**, start the emulation, and enable **HSFZ** or **DoIP** in Services. Keep the vehicle's transport profile at **ISO-TP (Classic)**. The service settings belong to your Studio workspace and are not enabled by opening a vehicle JSON file.

Demo	ECU request = reply addresses	Functional address	Tester	TCP / UDP
BMW HSFZ	Gateway 0x10; body 0x40; powertrain 0x12	0xDF	0xF4	6801 / 6811
Audi MDG1 DoIP	Powertrain 0x4076	0xE400	0x0E80	13400 / 13400

These arbitration fields represent logical network addresses on the internal bus. Do not replace them with CAN request/reply pairs such as 0x7E0/0x7E8. Every ECU also listens on its protocol's functional address. DoIP clients must activate routing before sending diagnostics; HSFZ has no DoIP routing handshake.

Connect a local tester to **127.0.0.1**; for a tester on another machine, use the emulator host's reachable IP and a service bind address that permits LAN connections. Read VIN with **22 F1 90**, software identification with **22 F1 01** (BMW) or **22 F1 87** (Audi), and send **3E 00** to the functional address to see all configured ECUs answer. Discovery and every ECU return the same synthetic VIN within each demo.

The installed directories are **/usr/local/cansole/share/specs/hsfz-demo/** and **/usr/local/cansole/share/specs/doip-demo/**. These are curated diagnostic demos based on existing TPE specs and CANcorder captures. The BMW body domain uses FEM_BODY data; it does not claim to reproduce a particular BDC generation. Flash routines, memory images and security algorithms are outside these demos.

K-Line demos on a remote ECUconnect

Studio includes **Demo Vehicles** → **K-Line** → **AL319 OBD-II** and **AL319 OBD-II · 2 ECUs**. Select the ECUconnect network runtime, load the demo and enable its K-Line service on onboard bus diagnostic. The two-ECU demo binds Engine at 0x10 and Transmission at 0x18. Both expose VIN/CALID/CVN, with distinct calibration identifiers. It also includes stored/pending DTCs and two Engine freeze frames associated with P0171 and P0420.

The vehicle-level kline.ecus list stores ECU names and physical addresses; existing OBD-II/KWP handlers hold the diagnostic data. The supported profile is 10400 baud, E9/8F keywords, Fast/Slow Init and up to 16 addressed ECUs. Functional requests can get replies from several ECUs. Compatible data edits preserve the physical connection; line-profile changes require initialization.

See [K-Line on ECUconnect](#) for setup, editing, status and limitations. Physical AL319 tests confirm stored and pending codes. Permanent-code support is tested in software; the AL319 manual limits that function to CAN. The first freeze frame passed on AL319; the trace contains no request for the second frame.

Next: [Connect Your Tester](#) explains how to expose the chosen demo, and [Writing Vehicle and ECU Specs](#) explains the BMW demo files in detail.

Example Spec Files

These are the complete BMW Motorrad demo files that [Writing Vehicle and ECU Specs](#) walks through. They are the files behind **Demo Vehicles** → **BMW Motorrad — 1250 Boxer** in the CAN group, included here unchanged. Copy their contents exactly and keep the file names: the vehicle file refers to the two handler files by name.

File	Content
bmw-motorrad-vehicle.json	Vehicle name, description, VIN and the two handlers of the BMS-X ECU
bmw-motorrad-uds.json	UDS handler: addresses, timing, identification DIDs and fault memory
bmw-motorrad-obd2.json	OBD-II handler: live PIDs, freeze frame, VIN information, fault memory and monitor results

On macOS, the installer also places these files in `/usr/local/cansole/share/specs/motorrad/`.

bmw-motorrad-vehicle.json

```
{
  "name": "BMW Motorrad — 1250 Boxer",
  "info": "Educational BMW Motorrad BMS-X example for building and testing a diagnostic client against one physical ECU with both UDS and OBD-II. Curated by Vanille Media.\n\nEmulates identification DIDs, live and freeze-frame OBD-II data, readiness and monitor results, VIN information, stored and permanent DTCs, and values that change through generators or sequences. Both handlers share CAN request 0x7E0 and reply 0x7E8; OBD-II also listens on 0x7DF. The authoring guide uses this vehicle step by step.\n\nUses a synthetic VIN and illustrative values. It does not reproduce the complete behavior of a real motorcycle or implement its flash programming and security algorithms.",
  "vin": "WB10G4105T6D00001",
  "ecus": [
    {
      "name": "BMS-X",
      "path": "bmw-motorrad-uds.json"
    },
    {
      "name": "BMS-X",
      "path": "bmw-motorrad-obd2.json"
    }
  ]
}
```

bmw-motorrad-uds.json

```
{
  "name": "BMW Motorrad Engine",
  "kind": "uds",
  "uds": {
    "arbitrationInfo": {
      "request": "0x7E0",
      "reply": "0x7E8",
      "auxiliary": []
    },
    "p2ServerMax": 0.05,
    "p2extServerMax": 2.0,
    "s3SessionTimeout": 5.0,
    "identifiers": [
      {
        "id": "0xF190",
        "name": "VIN",
        "type": "ascii",
        "content": "WB10G4105T6D00001"
      },
      {
        "id": "0xF187",
        "name": "Part Number",
        "type": "ascii",
        "content": "13618555123"
      },
      {
        "id": "0xF191",
        "name": "Hardware Version",
        "type": "ascii",
        "content": "BMSX-HW-03"
      },
      {
        "id": "0xF195",
        "name": "Software Version",
        "type": "ascii",
        "content": "BMSX-SW-0421"
      },
      {
        "id": "0xF180",
        "name": "Boot Software Identification",
        "type": "ascii",
        "content": "BMSX-B00T-1.04"
      },
      {
        "id": "0xF181",
        "name": "Application Software Identification",
        "type": "ascii",
        "content": "BMSX-APP-4.21"
      },
      {
        "id": "0xF182",
        "name": "Application Data Identification",
        "type": "ascii",
        "content": "BMSX-CAL-2026-A"
      },
      {
        "id": "0xF18C",
        "name": "ECU Serial Number",
        "type": "ascii",
        "content": "BMSX-26-0000001"
      },
      {
        "id": "0xF197",
        "name": "System Name",
        "type": "ascii",

```

```

        "content": "BMS-X ENGINE MGT"
    },
    {
        "id": "0xF19E",
        "name": "Configuration Identification",
        "type": "ascii",
        "content": "R1250-EU5-A"
    },
    {
        "id": "0xF100",
        "name": "Engine Speed",
        "comment": "Demo-defined: unsigned big-endian raw / 4 = 2000 rpm; same value as OBD PID 0C.",
        "bytes": "1F 40"
    },
    {
        "id": "0xF101",
        "name": "Coolant Temperature",
        "comment": "Demo-defined: raw - 40 = 90 degrees C; same value as OBD PID 05.",
        "bytes": "82"
    },
    {
        "id": "0xF102",
        "name": "Vehicle Speed",
        "comment": "Demo-defined: raw = 50 km/h; same value as OBD PID 0D.",
        "bytes": "32"
    },
    {
        "id": "0xF103",
        "name": "Module Voltage",
        "comment": "Demo-defined: unsigned big-endian raw / 1000 = 14 V; same value as OBD PID 42.",
        "bytes": "36 B0"
    },
    {
        "id": "0xF104",
        "name": "Status Bits",
        "comment": "Demo-defined bit mask: bits 0 and 2 set.",
        "bytes": "05"
    },
    {
        "id": "0xF105",
        "name": "Counter",
        "comment": "Demo-defined unsigned big-endian 32-bit counter: 123456.",
        "bytes": "00 01 E2 40"
    }
],
"securityLevels": [],
"routines": [],
"memoryLayout": [],
"dtcs": [
    {
        "bytes": [1, 48, 0],
        "comment": "P0130 oxygen sensor circuit, bank 1 sensor 1. Status 0x89: failed, confirmed, warning indicator requested.",
        "status": [
            "testFailed",
            "confirmedDTC",
            "warningIndicatorRequested"
        ]
    },
    {
        "bytes": [2, 1, 0],
        "comment": "P0201 injector circuit, cylinder 1. Status 0x09: failed and confirmed.",
        "status": [
            "testFailed",
            "confirmedDTC"
        ]
    }
],
{

```

```
    "bytes": [1, 23, 0],
    "comment": "P0117 coolant temperature sensor circuit low. Status 0x06: failed this cycle, pending.",
    "status": [
        "testFailedThisOperationCycle",
        "pendingDTC"
    ]
},
{
    "bytes": [21, 32, 19],
    "comment": "P1520 gear shift assistant sensor, circuit open (failure type 0x13). Manufacturer specific and
not emissions relevant, so it exists only here: no OBD-II mode may report it and mode 04 may not clear it. This is
the deeper fault layer a generic scan tool never sees. Status 0x09: failed and confirmed, with no warning indicator,
because the MIL is an emissions lamp.",
    "status": [
        "testFailed",
        "confirmedDTC"
    ]
}
],
"startupActions": []
}
```

bmw-motorrad-obd2.json

```
{
  "name": "BMW Motorrad Engine",
  "kind": "obd2",
  "obd2": {
    "arbitrationInfo": {
      "request": "0x7E0",
      "reply": "0x7E8",
      "auxiliary": [
        "0x7DF"
      ]
    },
    "vin": "WB10G4105T6D00001",
    "frames": [
      {
        "pids": [
          {
            "id": "0x01",
            "comment": "Monitor status: MIL on, two stored DTCs. Readiness claims only what a motorcycle carries: catalyst, oxygen sensor and its heater. The catalyst monitor is incomplete, which is why the permanent DTC cannot clear yet. Fixed sample bytes; mode 04 does not recompute them.",
            "simple": "82 07 61 01"
          },
          {
            "id": "0x03",
            "comment": "Fuel system 1 closed loop.",
            "simple": "02 00"
          },
          {
            "id": "0x04",
            "comment": "Engine load:  $A * 100 / 255 =$  approximately 50.2%.",
            "simple": "80"
          },
          {
            "id": "0x05",
            "comment": "Coolant:  $A - 40 =$  90 degrees C.",
            "simple": "82"
          },
          {
            "id": "0x06",
            "comment": "Short-term fuel trim:  $(A - 128) * 100 / 128 =$  0%.",
            "simple": "80"
          },
          {
            "id": "0x07",
            "comment": "Long-term fuel trim: 0%.",
            "simple": "80"
          },
          {
            "id": "0x0A",
            "comment": "Fuel pressure:  $A * 3 =$  300 kPa.",
            "simple": "64"
          },
          {
            "id": "0x0B",
            "comment": "Manifold pressure:  $A =$  100 kPa.",
            "simple": "64"
          },
          {
            "id": "0x0C",
            "comment": "Engine speed:  $(256*A + B) / 4 =$  2000 rpm.",
            "simple": "1F 40"
          },
          {
            "id": "0x0D",
            "comment": "Vehicle speed:  $A =$  50 km/h.",

```

```

    "simple": "32"
  },
  {
    "id": "0x0E",
    "comment": "Timing advance: A / 2 - 64 = 0 degrees.",
    "simple": "80"
  },
  {
    "id": "0x0F",
    "comment": "Intake air: A - 40 = 25 degrees C.",
    "simple": "41"
  },
  {
    "id": "0x11",
    "comment": "Throttle: A * 100 / 255 = approximately 25.1%.",
    "simple": "40"
  },
  {
    "id": "0x1C",
    "comment": "Compliance: EOBD. A machine sold in the EU is not CARB certified.",
    "simple": "06"
  },
  {
    "id": "0x33",
    "comment": "Barometric pressure: A = 100 kPa.",
    "simple": "64"
  },
  {
    "id": "0x42",
    "comment": "Module voltage: (256*A + B) / 1000 = 14 V.",
    "simple": "36 B0"
  },
  {
    "id": "0x46",
    "comment": "Ambient temperature: A - 40 = 20 degrees C.",
    "simple": "3C"
  },
  {
    "id": "0x51",
    "comment": "Fuel type: gasoline.",
    "simple": "01"
  },
  {
    "id": "0x1F",
    "comment": "Successive reads return 10, 20, 30 seconds and repeat; not elapsed time.",
    "sequence": {
      "responses": [
        "00 0A",
        "00 14",
        "00 1E"
      ],
      "repeat": "loop"
    }
  },
  {
    "id": "0x2F",
    "comment": "Tank level: 25-75 percent over 60 seconds. Raw = percentage * 2.55.",
    "dynamic": {
      "function": "sawtooth",
      "min": 25,
      "max": 75,
      "period": 60,
      "encoding": {
        "type": "uint8",
        "scale": 2.55
      }
    }
  }
},

```

```

{
  "id": "0x14",
  "comment": "02 sensor 1 voltage:  $A / 200 = 0.45$  V, B = 128 means no trim applied.",
  "simple": "5A 80"
},
{
  "id": "0x21",
  "comment": "Distance with MIL on: 0 km, consistent with the MIL-off monitor status.",
  "simple": "00 00"
},
{
  "id": "0x30",
  "comment": "Warm-ups since codes cleared: 40.",
  "simple": "28"
},
{
  "id": "0x31",
  "comment": "Distance since codes cleared:  $256 * A + B = 8000$  km.",
  "simple": "1F 40"
},
{
  "id": "0x34",
  "comment": "02 sensor 1 wide range: equivalence ratio 1.0, current 0 mA.",
  "simple": "80 00 80 00"
},
{
  "id": "0x44",
  "comment": "Commanded equivalence ratio:  $(256 * A + B) / 32768 = 1.0$ .",
  "simple": "80 00"
},
{
  "id": "0x45",
  "comment": "Relative throttle position:  $A * 100 / 255 =$  approximately 10.2%.",
  "simple": "1A"
},
{
  "id": "0x49",
  "comment": "Throttle grip sensor 1:  $A * 100 / 255 =$  approximately 25.1%.",
  "simple": "40"
},
{
  "id": "0x4A",
  "comment": "Throttle grip sensor 2: the redundant channel reports the same angle.",
  "simple": "40"
},
{
  "id": "0x4C",
  "comment": "Commanded throttle actuator:  $A * 100 / 255 =$  approximately 25.1%.",
  "simple": "40"
},
{
  "id": "0x5C",
  "comment": "Engine oil temperature:  $A - 40 = 70$  degrees C.",
  "simple": "6E"
},
{
  "id": "0x5E",
  "comment": "Engine fuel rate:  $(256 * A + B) / 20 = 4.0$  L/h.",
  "simple": "00 50"
},
{
  "id": "0xF1",
  "comment": "Custom lab PID, not a standard measurement: unsigned big-endian demo value 4660. Tester must know this definition.",
  "simple": "12 34"
},
{
  "id": "0xF2",

```

```

        "comment": "Custom lab PID: four ASCII bytes spelling DEMO. Tester must know this definition.",
        "simple": "44 45 4D 4F"
    },
    {
        "id": "0xF3",
        "comment": "Custom lab PID: one-byte status sequence 0, 1, 2, then repeat.",
        "sequence": {
            "responses": [
                "00",
                "01",
                "02"
            ],
            "repeat": "loop"
        }
    }
]
},
{
    "pids": [
        {
            "id": "0x05",
            "comment": "Freeze-frame coolant: 50 degrees C.",
            "simple": "5A"
        },
        {
            "id": "0x0C",
            "comment": "Freeze-frame engine speed: 1000 rpm.",
            "simple": "0F A0"
        }
    ]
}
],
"informational": [
    {
        "id": "0x02",
        "comment": "One item (01), followed by 17 VIN ASCII bytes. Keep consistent with UDS VIN.",
        "simple": "01 57 42 31 30 47 34 31 30 35 54 36 44 30 30 30 30 31"
    },
    {
        "id": "0x04",
        "comment": "Mode 09 calibration ID: one item (01), followed by a 16-byte ASCII ID.",
        "simple": "01 42 4D 53 58 43 41 4C 2D 52 31 32 35 30 2D 32 36"
    },
    {
        "id": "0x06",
        "comment": "Mode 09 calibration verification number: one item, then four demonstration bytes. Not a computed checksum.",
        "simple": "01 12 34 56 78"
    },
    {
        "id": "0x0A",
        "comment": "Mode 09 ECU name; demonstration ASCII text.",
        "simple": "01 42 4D 53 00 2D 58 20 45 4E 47 49 4E 45 20 4D 47 54 00 00 00"
    }
],
"oxygenSensorMonitoring": [],
"onBoardMonitoring": [
    {
        "id": "0x01",
        "comment": "O2 sensor bank 1 sensor 1, rich to lean threshold. TID 01, unit 0B (30.5 uV per bit), then value, minimum and maximum. 0.244 V is below the 0.350 V minimum, which is the measurement behind P0130.",
        "simple": "01 0B 1F 40 2C D3 46 71"
    },
    {
        "id": "0x02",
        "comment": "O2 sensor bank 1 sensor 1, lean to rich threshold. Same encoding; 0.360 V lies inside its limits.",
        "simple": "02 0B 2E 1B 23 28 39 A2"
    }
]

```

```

    },
    {
      "id": "0x21",
      "comment": "Catalyst monitor bank 1. TID 81 is manufacturer defined, unit 01 is a raw count. Demonstration
values.",
      "simple": "81 01 01 90 00 00 02 58"
    },
    {
      "id": "0xA2",
      "comment": "Misfire counts, cylinder 1 of the boxer twin. Zero counted against a limit of 100.",
      "simple": "0B 24 00 00 00 00 00 64"
    },
    {
      "id": "0xA3",
      "comment": "Misfire counts, cylinder 2. Same encoding.",
      "simple": "0B 24 00 00 00 00 00 64"
    }
  ],
  "controlOperations": [],
  "dtcs": [
    {
      "bytes": [1, 48, 0],
      "comment": "P0130 oxygen sensor circuit, bank 1 sensor 1. Confirmed, MIL requested.",
      "status": [
        "testFailed",
        "confirmedDTC",
        "warningIndicatorRequested"
      ]
    },
    {
      "bytes": [2, 1, 0],
      "comment": "P0201 injector circuit, cylinder 1. Confirmed.",
      "status": [
        "testFailed",
        "confirmedDTC"
      ]
    },
    {
      "bytes": [1, 23, 0],
      "comment": "P0117 coolant temperature sensor circuit low. Pending only, so it appears in mode 07 and not in
mode 03.",
      "status": [
        "testFailedThisOperationCycle",
        "pendingDTC"
      ]
    }
  ],
  "permanentDtcs": [
    {
      "bytes": [1, 48, 0],
      "comment": "A permanent DTC survives mode 04 by design. It clears only once its monitor passes, which is
what makes the clear behaviour testable.",
      "status": [
        "confirmedDTC"
      ]
    }
  ]
}

```

Troubleshooting

This chapter collects problems that are not specific to one feature. The chapters on individual services and on spec authoring have their own checks:

Area	Where to look
JSON files, missing replies, wrong values	Edit, Apply and Retest , Writing Vehicle and ECU Specs
Physical CAN adapters	External CAN
Ethernet testers	HSFZ , DoIP
OBD-II apps	ELM327/STN
Remote vehicle sessions	ENET Service Broker
K-Line testers	K-Line on ECUconnect
Reading a vehicle	Read from Vehicle

General checks

Symptom	First checks
A local tester cannot connect	Connect to 127.0.0.1, not localhost: localhost can resolve to the IPv6 address ::1, while the services listen on IPv4
A tester on another computer cannot connect	Use the reachable IP address of the emulator host, a bind address that permits LAN connections, and check the firewall
No service works	The Internal Bus must be enabled; all other services depend on it
HSFZ, DoIP or ELM327 is unavailable	These services need a classic transport profile. They are unavailable while the vehicle uses isotp_fd
Applying a vehicle or enabling an ECU is refused	Check the license under Help → License & Updates... . A network runtime needs a license on its own host
A feature is missing on a network runtime	Some features, such as reading a vehicle, need a current ECUlator++ runtime on that host
An open demo lacks new data	Demos are copied into the editor when loaded. Reload the demo from the menu; this discards unsaved demo edits
An edit has no effect	Click Apply Spec and check which runtime Studio is connected to
Open PDF manual reports a missing file	Reinstall Studio. Development builds generate the manual with <code>npm run build:manuals</code>

Network runtimes

When Studio controls a runtime on another computer or on an ECUconnect adapter, several things belong to that host rather than to your computer:

- CAN adapters and their names are those of the runtime host.
- CAN traces are written on the runtime host. Start that runtime with `--can-log` to record; Studio can only reveal traces of a managed local runtime.
- The license is checked on the runtime host. Connecting Studio does not transfer your license.

Reporting a problem

Note the Studio version from **Help** → **About ECUlator Studio** and, for a network runtime, the output of `ecum --version` on its host. A CAN trace (see [Edit, Apply and Retest](#)) and the vehicle files reproduce most emulation problems.