



USER GUIDE

CalTool User Guide

Contents

1. Copyright Notice	6
2. Introduction	7
2.1. The figures	7
2.2. Symbols Description	7
2.3. Contact Information	8
3. Getting Started	9
3.1. System requirements	9
3.2. Installing	9
3.3. The main window	9
3.4. The activity bar	10
3.5. The status bar	10
3.6. Settings	11
3.7. Keyboard shortcuts	11
4. Licensing	13
4.1. Activating CalTool	13
4.2. License status and renewal	13
4.3. Moving a license to another computer	14
5. Adding ECUs and Connecting	15
5.1. The device list	15
5.2. Connection settings	15
5.3. Going online	16
5.4. Calibration pages	17
5.5. Listening to a CAN bus	17
6. Workspace Files	18
7. Building a Workspace Page	19
7.1. The Explorer	19
7.2. Cards	20
7.3. The context menu	21
8. Calibrating	22
8.1. Editing tables	22
8.2. Change marks	22
8.3. Loading one value from a file	23
8.4. The working point	23
8.5. Single values	23
9. Logging	24
9.1. Choosing what is logged	24
9.2. Recording	25
9.3. Bars, gauges and highlights	26
9.4. Alarm limits	27
9.5. Measurements you can set	27
9.6. Saving a log	28

9.7. Opening a log	28
9.8. Exporting to MATLAB or CSV	29
10. Browsing Logs	30
10.1. The log table	30
10.2. Showing part of a log	31
10.3. Comparing and combining logs	32
11. Plotting	34
11.1. Cursors	35
11.2. A plot as a picture	35
11.3. Gaps in a recording	35
11.4. Time across plots	35
11.5. Subplots	35
11.6. A signal as the X axis	35
11.7. Between the cursors	36
11.8. Markers	36
12. Video and maps	38
12.1. Adding a video	38
12.2. The video card	39
12.3. Lining a video up with the log	40
12.4. Recording video with a log	40
12.5. Recording with a GoPro	40
12.6. GoPro signals	41
12.7. The map card	41
12.8. Maps without internet	42
12.9. Saving a log with its video	42
13. Working with an AI assistant	43
13.1. Switching AI access on	43
13.2. What the assistant may do	45
13.3. Connecting an AI tool	46
13.4. Working with the assistant	47
13.5. If the assistant cannot reach CalTool	48
14. Calibration Data	49
14.1. Loading and saving	49
14.2. Comparing calibration data	49
15. CAN Trace	51
15.1. Watching a bus	51
15.2. Sending messages	52
15.3. Replaying a log	52
15.4. CAN logs and the measurement log	53
16. Flashing and Diagnostics	54
16.1. Flashing over XCP	54
16.2. Diagnostics over UDS and KWP2000	54
Appendix A: ECU Description Files (.uds)	57

A.1. Example	57
A.2. How a file is written	59
A.3. Connection and session settings	60
A.4. Steps	60
A.5. Actions	65
A.6. Examples	66
A.7. KWP2000 ECUs	72
A.8. Security access	73
A.9. Data identifiers	73
A.10. Fault code texts	74
A.11. The flash sequence	74
A.12. Writing a file for a new ECU	76
Appendix B: Computed Channels	77
B.1. Creating a computed channel	77
B.2. How the script works	78
B.3. Examples	78
Appendix C: CAN Stimulation: Counters, Checksums and Scripts	81
C.1. The example messages	81
C.2. Counters and checksums in the transmit panel	81
C.3. Scripts	83

Document Information

Document Number	100-0-0004
Created By	Alexander Olsson
Date	2026-10-19
Revision	Rev. 00.06
Classification	EXTERNAL
Applies to	Altech CalTool (ECU calibration, measurement and diagnostics tool)

Revision History

Revision	Date	Author	Change
00.01	2026-09-14	Alexander Olsson	Initial release
00.02	2026-09-23	Alexander Olsson	Updated for the new user interface and licensing. New figures throughout; text shortened.
00.03	2026-09-28	Alexander Olsson	Added Appendix C: counters, checksums and scripts for CAN stimulation. Added chapter Video and maps.
00.04	2026-10-08	Alexander Olsson	Added chapter Browsing Logs (the Logs pane). CAN FD, CAN log replay and CAN traffic in measurement logs, recording without an ECU, DAQ overload warnings, subplots, time sync, X-Y plots, cursor statistics, log time range, INCA .ee import, KWP2000 diagnostics and confirm steps. Markers in plots and logs. Comparing and combining logs. Every common UDS service as a named step in ECU description files, with examples. New activity bar and status bar figures.
00.05	2026-10-16	Alexander Olsson	Added chapter Working with an AI assistant: switching AI access on, permissions and the confirmation of ECU writes, connecting Claude Code, Claude Desktop, ChatGPT and other tools, and what to ask.
00.06	2026-10-19	Alexander Olsson	Full review against the current release. Added A2L file warnings, ECU messages, reconnecting during a recording, dials and alarm limits, writable measurements, reverting changes, loading one value from a file, DCM files, plot cursors and their keys, plots as pictures, and the menu entries that changed. New measurement card figure.

1. Copyright Notice

This document is the property of Altech Electronics Nordic AB. The document may not be edited, utilized, reproduced or otherwise exploited, in full or in part, without the written permission of Altech Electronics Nordic AB.

2. Introduction

Altech CalTool is a Windows application for calibrating, measuring and diagnosing ECUs:

- **Connect** to several ECUs over XCP on CAN, CAN FD or Ethernet, and to CAN buses through their DBC files, in one workspace.
- **Calibrate** values, curves, maps and arrays in tables, charts and 3D, with the ECU's live working point.
- **Measure** with live readouts, bars, dials and alarm limits.
- **Log** with XCP DAQ to MDF4, together with CAN traffic and video, and analyse recordings with plots, cursors, markers, readouts and computed channels. Compare runs. Export to MATLAB or CSV.
- **Browse logs**: a folder of logs as a table, with their descriptions.
- **Manage calibration data**: **.cdfx** and **.dcm** files, memory images and side-by-side comparison.
- **Work with the CAN bus**: trace, decode, log, transmit, replay and script, on classic CAN and CAN FD.
- **Diagnose and flash** over UDS, KWP2000, XCP or the Vector VX flash tool.
- **Work with an AI assistant**: Claude, ChatGPT and other AI tools connect to your open session to analyse logs, build computed channels, mark findings and—with your permission and confirmation—calibrate ([Chapter 13](#)).

Support for further protocols, interfaces and file formats can be added on request; contact Altech.

Hover over a control for a tooltip, and right-click almost anything for the options that apply to it. Numbered markers in the figures match the numbered list below each figure.

2.1. The figures

The figures were made with a small demonstration ECU and CAN database on CalTool's built-in **Simulation** channel—a virtual ECU and CAN bus, so every step can be repeated without hardware. They contain no customer data. Where a figure shows a value, a file name or a description, it is an example.

2.2. Symbols Description

The following symbols are used in this user guide to highlight certain information.



Critical information—the requirements must be followed in order to not affect the performance of the system or product.



Recommended information—the requirements should be followed in order to not

affect the performance of the system or product.



Additional information — further information can be found in a referenced Altech guideline or datasheet. It's recommended to study all referenced documents before installing any Altech system or product.

2.3. Contact Information

For information or sales inquiries, please contact info@altechelectronics.io.

3. Getting Started

3.1. System requirements

Item	Requirement
Operating system	Windows 10 or 11, 64-bit
CAN interface	Vector, Kvaser or PEAK with its driver installed. Not needed for Simulation, XCP on Ethernet or log analysis.
Network	Internet for license activation, and at least once every 30 days (Chapter 4)

3.2. Installing

Run **CalTool-Setup-<version>.exe** (administrator rights needed). CalTool appears in the Start menu under **Altech**. On first start it asks for your license key ([Section 4.1](#)).

When CalTool starts, it reopens the workspace that was open when it was last closed.

3.3. The main window

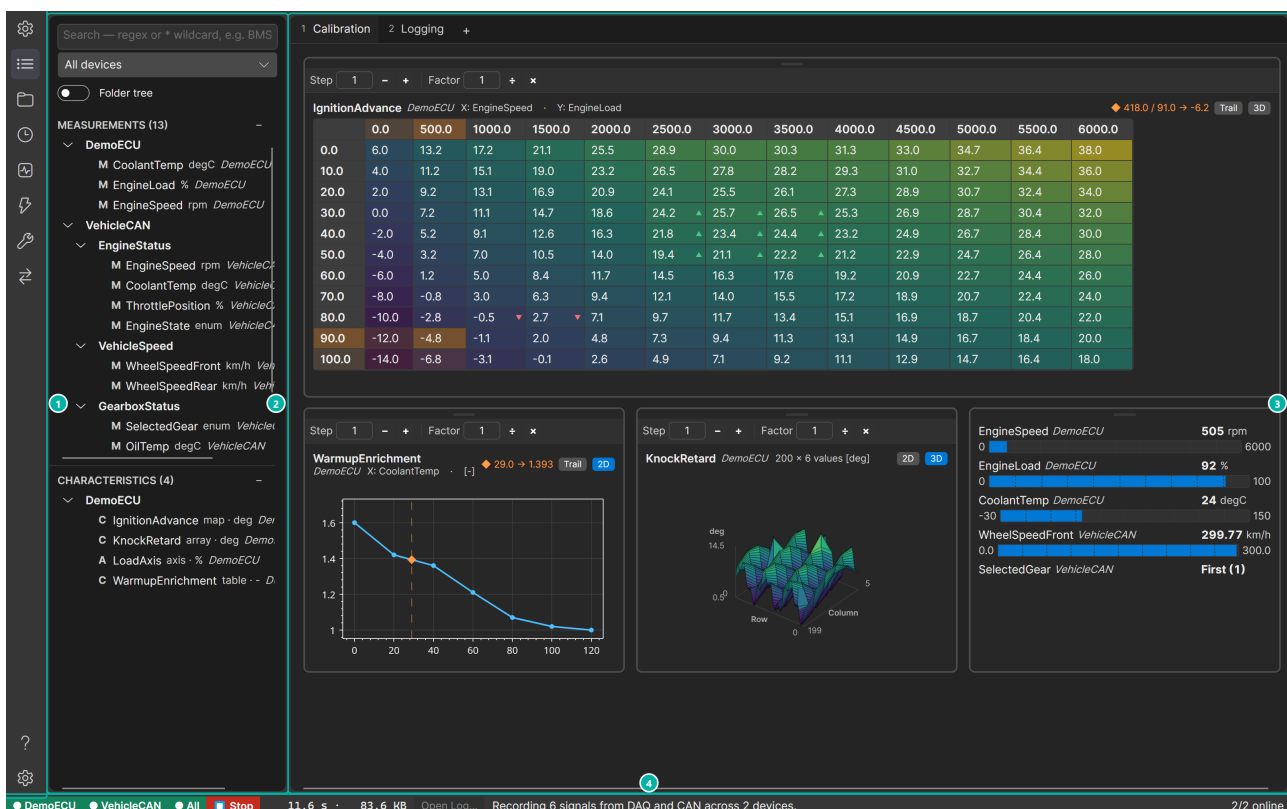
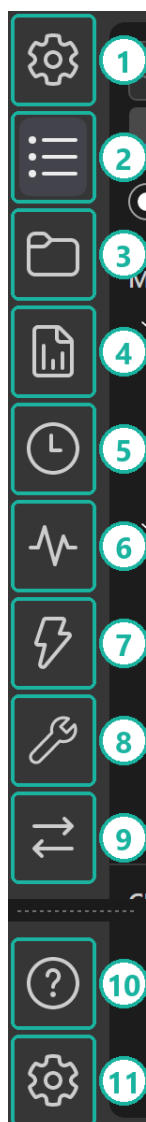


Figure 1. The main window

1. **Activity bar** — selects what the side bar shows.
2. **Side bar** — the selected pane. Drag its edge to resize it.
3. **Workspace** — pages of cards: measurements, calibration values and plots.

4. **Status bar** — devices, recording and messages.

3.4. The activity bar



1. **ECU Setup** — devices and connections ([Chapter 5](#))
2. **Measurements & Characteristics** — the Explorer ([Section 7.1](#))
3. **Files & Calibration Data** — workspaces, video, calibration files ([Chapter 6](#))
4. **Logs** — browse, open and save logs ([Chapter 10](#))
5. **Measurement Rasters** — what is logged ([Chapter 9](#))
6. **CAN Trace** — raw CAN traffic ([Chapter 15](#))
7. **Flash Firmware (XCP)** — flashing ([Section 16.1](#))
8. **Diagnostics (UDS)** — fault codes and diagnostics ([Section 16.2](#))
9. **Compare Calibration Data** — compare calibration files ([Section 14.2](#))
10. **User guide** — opens this document
11. **Settings** — theme, text size and license

3.5. The status bar

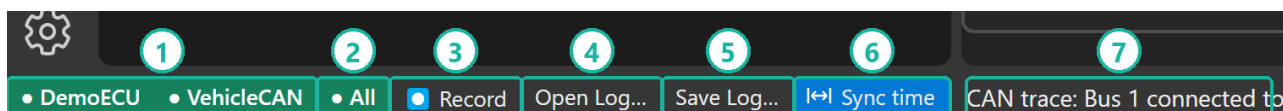


Figure 2. The status bar, left end

1. **Device chips** — green: online, red: connection failed. Click to go online or offline.
2. **All** — all devices online or offline.
3. **Record** — starts and stops recording ([Section 9.2](#)). While recording, the elapsed time and a warning if the ECU cannot keep up.
4. **Open Log...** — opens a log file.

5. **Save Log...** — appears after a recording.
6. **Sync time** — zooming or panning one plot moves every plot (Section 11.4). On by default.
7. **Status message** — the result of the last action.

These appear at the right end when they need your attention:

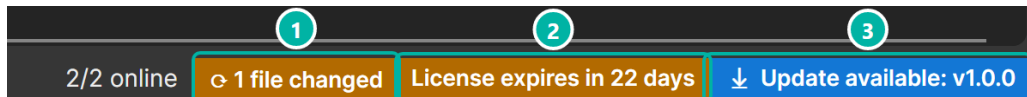


Figure 3. Notices at the right end of the status bar

1. **File changed** — a loaded A2L, memory image or DBC file was changed by another program, for example a new software build. Click to reload it.
2. **License** — the license expires soon (amber) or cannot be used (red). Click to open the License dialog (Chapter 4).
3. **Update available** — click to download and install the new version. Settings, license and workspaces are kept.

3.6. Settings

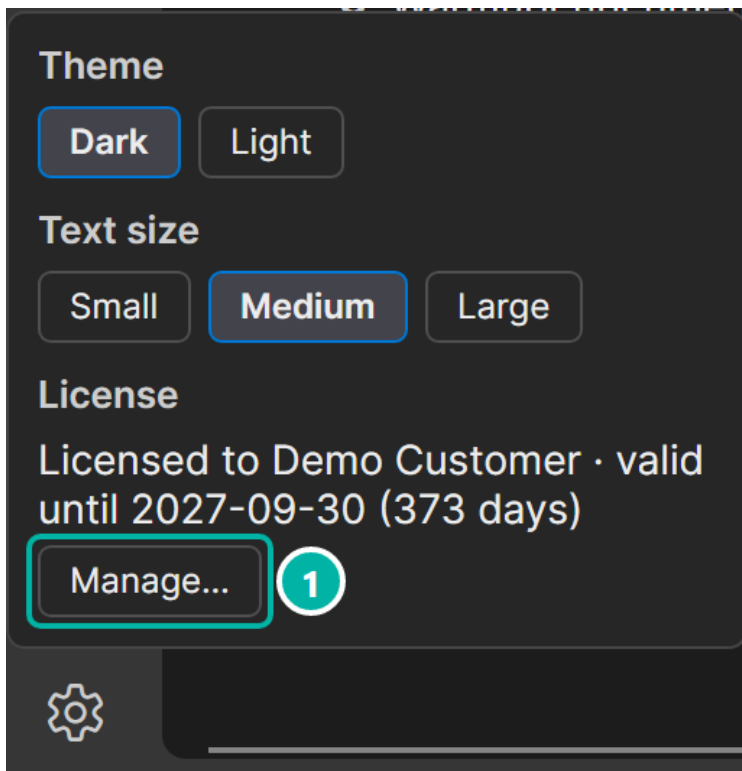


Figure 4. The settings menu

Click the gear at the bottom of the activity bar to choose the theme and text size. The light theme is made for bright conditions, such as a laptop in a car.

1. **Manage...** — opens the License dialog.

3.7. Keyboard shortcuts

Key	Action
S	Start recording / stop and save the log

Key	Action
Shift + S	Stop recording without saving yet
M	Add a marker — now while recording, else at the cursor (Section 11.8)
Alt + 1 ... Alt + 9	Go to page 1 to 9 (keep Alt held and type more digits for page 10 and up)
Ctrl + B	Hide or show the side bar
Ctrl + C	Copy the names of the selected rows
Ctrl + ← / Ctrl + →	In a plot: step the cursor one sample (Section 11.1)
Delete	Remove the selected rows from a list or plot

Shortcuts are ignored while typing in a text field. Tables have their own keys ([Section 8.1](#)).

4. Licensing

CalTool is licensed per computer for a fixed period, with a license key from Altech. A key is active on one computer at a time and can be moved ([Section 4.3](#)).

4.1. Activating CalTool

The first time CalTool starts, it asks for your license key.

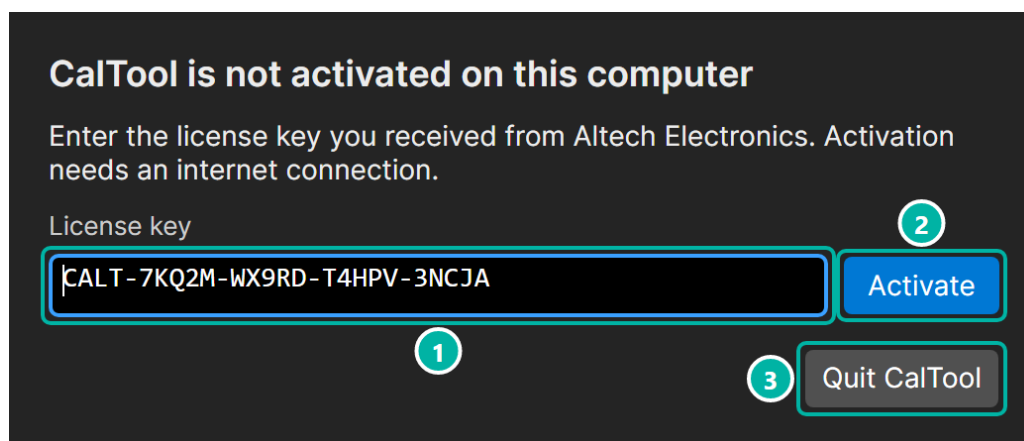


Figure 5. Activating CalTool

1. **License key** — type or paste the key, **CALT-XXXXX-XXXXX-XXXXX-XXXXX**.
2. **Activate** — needs an internet connection.
3. **Quit CalTool** — closes CalTool without activating.



If activation fails behind a company firewall, ask your IT department to allow HTTPS to license.altechelectronics.io.

4.2. License status and renewal

Open the License dialog from **Settings > Manage...**, or by clicking the license notice in the status bar ([Section 3.5](#)).

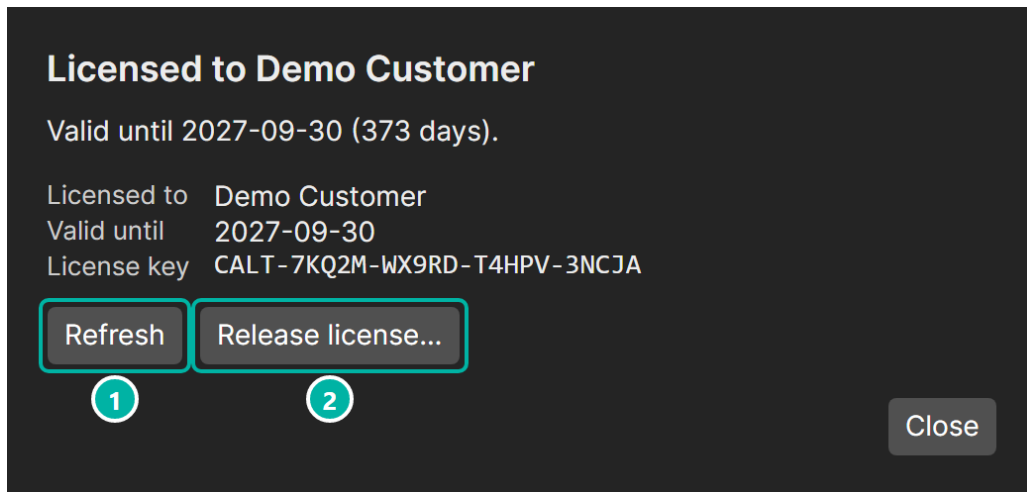


Figure 6. The License dialog

1. **Refresh** — checks the license with Altech now. Use it after a renewal.
2. **Release license...** — frees the license for use on another computer.

From 30 days before expiry, the status bar shows how many days are left. Contact Altech to renew; the renewal reaches CalTool automatically, or at once with **Refresh**.

CalTool works offline, but must reach the license server at least once every 30 days. Otherwise it shows **License check overdue** until it is connected to the internet again and refreshed.

If the license cannot be used, for example because it has expired, the License dialog explains why and CalTool closes when the dialog is closed.

4.3. Moving a license to another computer

1. On the old computer: **Settings > Manage... > Release license...**, and click again to confirm.
2. On the new computer: activate the same key.

If the old computer is broken or lost, contact Altech.

5. Adding ECUs and Connecting

A **device** is an ECU, described by its A2L file, or a CAN bus that CalTool listens to, described by its DBC file. Devices are managed in the **ECU Setup** pane.

5.1. The device list

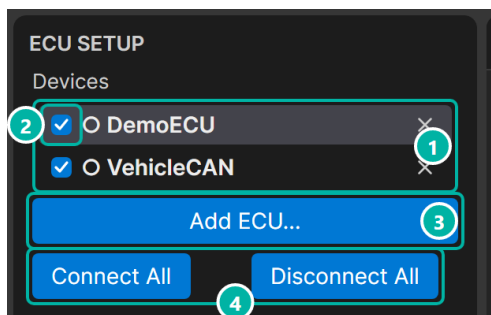


Figure 7. ECU Setup: the device list

1. **Device list** — drag to reorder; the cross removes a device (take it offline first).
2. **Include in Connect All** — untick to leave a device out.
3. **Add ECU...** — choose an **.a2l** file, or a **.dbc** file for a CAN bus (Section 5.5).
4. **Connect All / Disconnect All** — a device that fails is skipped; the others still connect.

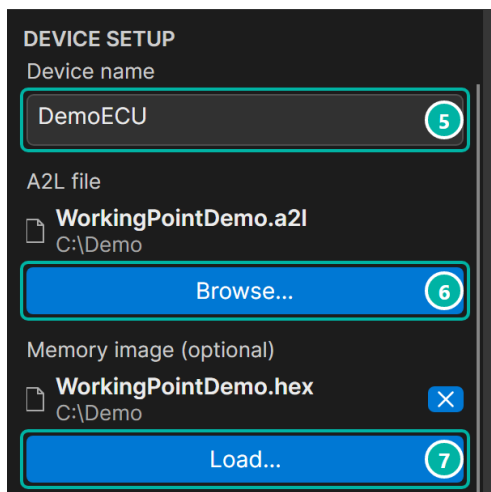


Figure 8. ECU Setup: device name and files

5. **Device name** — shown on the status bar chip and with every signal of the device.
6. **Browse...** — replaces the A2L file, for example after a new software build. Cards and values are kept.
7. **Memory image (optional)** — a **.hex** or **.s19** file for the ECU. Its values are imported, it is checked against the A2L file, and it is the reference for change marks (Section 8.2) and **Save to memory image** (Chapter 14).

Right-click a file name to copy its path or open its folder.

Warnings about the A2L file

An A2L file with errors — a damaged block, an unknown keyword, objects without an address — is loaded as far as it goes. A **n warnings** line appears under the device's status; click it to read the list, for example to send it to the supplier of the file. Objects at address 0 are usually symbols the software build did not include. A file that is not an A2L file at all is refused.

5.2. Connection settings

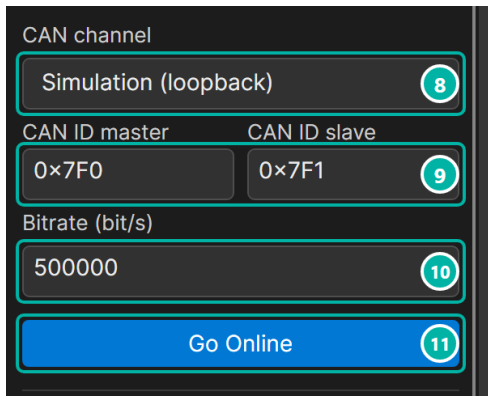


Figure 9. ECU Setup: connection settings

8. **CAN channel**—a Vector, Kvaser or PEAK interface, or **Simulation (loopback)**.
9. **CAN ID master / slave**—the XCP identifiers, from the A2L file.
10. **Bitrate**—from the A2L file.
11. **Go Online**—connects to the ECU.

Each device remembers its interface by serial number, so two identical interfaces are never mixed up. If the A2L file describes both CAN and Ethernet, a **Connection** selector lets you choose; Ethernet shows the IP address, port and protocol instead.

CAN FD: tick the box under the bitrate and enter the data bitrate and sample points. An ECU whose A2L file describes CAN FD has them filled in. The same box is under every bitrate field in CalTool (CAN Trace, Flashing, Diagnostics). An interface that does not support CAN FD shows a warning.

If two devices are set to the same interface, both say so; that is a warning, not a refusal, since several ECUs often share one bus. When another program or pane already holds the channel at a different bitrate, the device connects at that rate and says which.

5.3. Going online

Go Online connects, checks that the ECU's software matches the A2L file (EPK), and compares the ECU's calibration with the workspace. If the software does not match, CalTool stays offline and offers to flash the ECU ([Chapter 16](#)). If the calibration differs, it asks which side to keep:

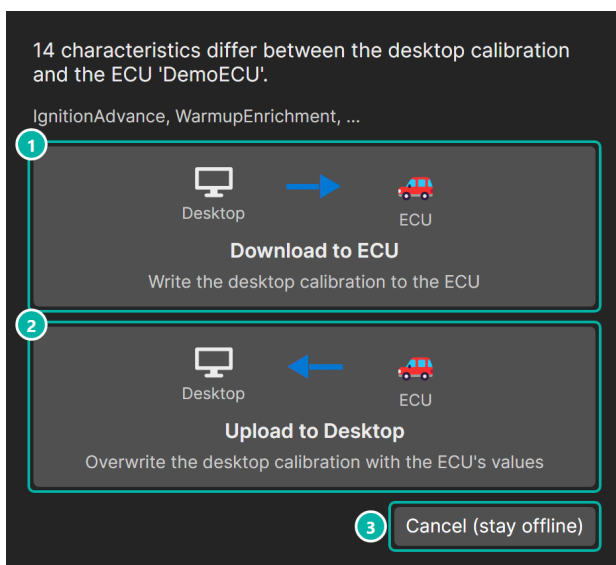


Figure 10. The calibration sync dialog

1. **Download to ECU**—write the workspace's values into the ECU.
2. **Upload to Desktop**—take the ECU's values into the workspace.
3. **Cancel (stay offline)**—change nothing.



While online, every calibration change is written to the ECU immediately. Make sure the ECU is in a state where calibration changes are safe.

Going online does not start measuring; see [Section 9.2](#). If an ECU stops responding, it goes offline and its chip turns red.

When the connection is lost during a recording

If an ECU stops answering while you are recording, the recording goes on with the other devices. CalTool marks the moment in the log, draws the ECU's last value as a dashed line in the plots, and tries to reconnect every two seconds — the chip and the button show **Reconnecting...** When the ECU answers again, its measurements continue in the same recording and the end of the gap is marked too. **Go Offline** stops trying.

Messages from the ECU

Some ECUs and measurement hardware send messages of their own, for example status texts or a warning that they cannot keep up. They are listed under the device's status as **n device messages**. A new warning puts **n** on the device's chip and in the list; opening the list clears it. **Copy** and **Clear** act on the list.

5.4. Calibration pages

If the ECU has several calibration pages, a **Calibration page** selector appears while online: **Working** (RAM, editable) or **Reference** (flash, read-only). While the reference page is selected, the calibration cards are greyed out.

If the ECU supports it, **Store working page in flash** keeps the calibration after the ECU is switched off. Do not switch the ECU off while it stores.

5.5. Listening to a CAN bus

A device added from a **.dbc** file decodes the bus traffic. Its signals appear in the Explorer, grouped by message, and can be shown, plotted and logged together with the ECUs' measurements (assign them to the **CAN** raster, [Chapter 9](#)).

6. Workspace Files

A **workspace file** (.ctws) saves the whole bench setup: devices, pages and cards, logging choices, calibration values, plots and the open log. Devices are not connected automatically when a workspace is opened.

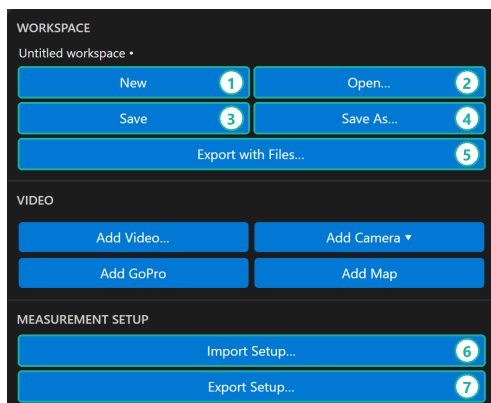


Figure 11. Files pane: workspace, video and measurement setup

1. **New** — an empty workspace.
2. **Open...** — opens a .ctws file.
3. **Save** — a dot (•) after the file name means unsaved changes.
4. **Save As...**
5. **Export with Files...** — a copy that includes the A2L, DBC, memory image and .uds files, so it opens on any computer. Logs are not included.
6. **Import Setup...** — sets up logging from an INCA .lab label list or an INCA experiment export (.ee): which measurements, in which raster. CalTool asks which device in the file is which ECU.
7. **Export Setup...** — writes the current logging setup as a .lab file.

If a file is missing when a workspace is opened, pick it again in ECU Setup; the values and settings are kept.

The pane also has **VIDEO** (Chapter 12), **CALIBRATION DATA** (Chapter 14) and **COMPUTED CHANNELS** (Appendix B). Log files are handled in the **Logs** pane (Chapter 10).

7. Building a Workspace Page

A workspace page is a canvas of **cards**. Drag objects from the Explorer onto a page to add them.

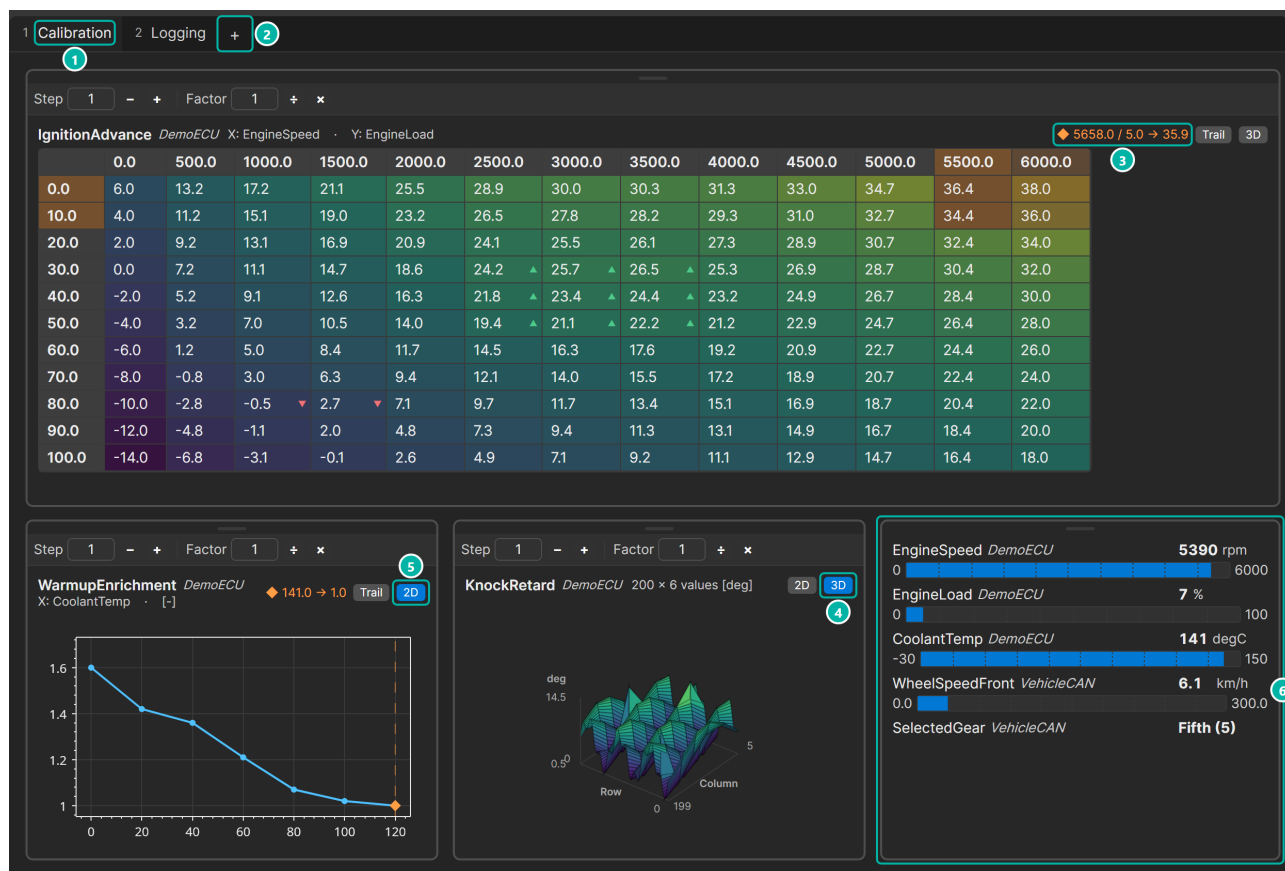


Figure 12. A calibration page: map table, curve chart, array in 3D and live measurements

- Page tab** — double-click to rename, drag to reorder.
- +** — adds a page.
- Working point** — where the ECU is operating on the map (Section 8.4).
- 3D** — shows a map or array as a rotatable surface.
- 2D** — shows a curve or array as a chart.
- Measurement list** — live values, here as bars (Section 9.3).

7.1. The Explorer

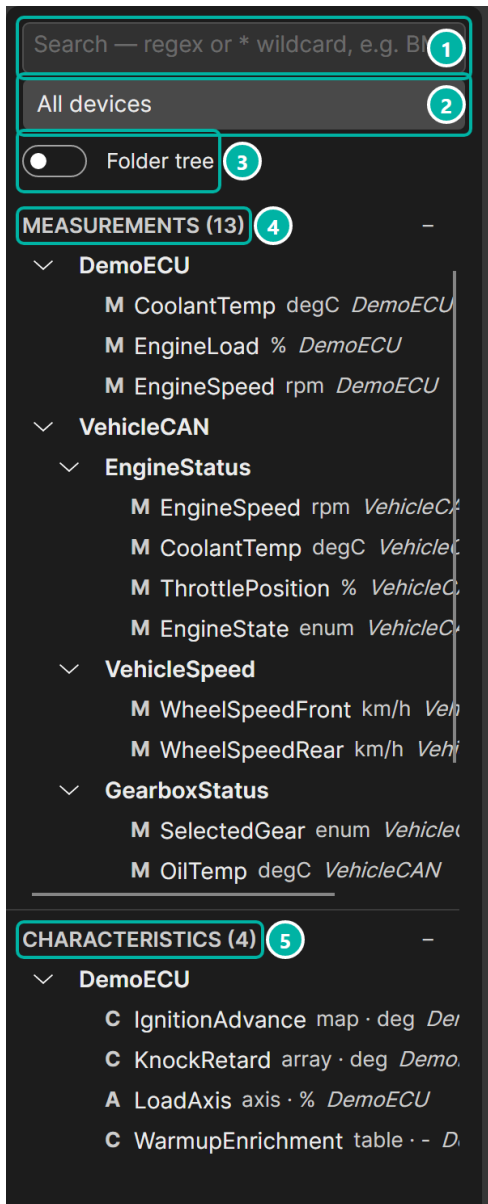


Figure 13. Explorer

1. **Search** — * is a wildcard; regular expressions work too.
2. **Device filter** — choose which devices are listed.
3. **Folder tree / Flat list** — grouped in folders, or alphabetical.
4. **MEASUREMENTS (M)** — values the ECU measures.
5. **CHARACTERISTICS (C, A)** — calibration values and axis tables.

Select several rows with **Ctrl** or **Shift** and drag them together. An open log adds a **LOG SIGNALS** section.

7.2. Cards

Dragged	Appears as
Measurements	A list card
Single calibration values	An editable list card
A curve, map, array or axis table	A table card, with chart or 3D view where it applies
Several tables	One card with a section per table
An array measurement	A dialog asks which elements to show

- **Move** a card by its thin top strip; **resize** from any edge. Cards snap to their neighbours (hold **Alt**)

to place freely).

- **Maximize** and **close** appear in the top-right corner when the pointer is over a card.
- **Combine** list cards, or table cards, by dropping one onto the middle of the other.
- **Move to another page** by dragging a card onto the page's tab.
- **Rename** a card by double-clicking its top strip.

7.3. The context menu

Right-click a measurement row for its options:

Entry	Does
Convert to plot	Turns the selected rows into a plot (Chapter 11).
Highlight	A background colour, to mark groups.
Display as	Decimal, hexadecimal or binary.
Bar display / Gauge display	Shows the value as a bar or a dial (Section 9.3).
Bar/gauge range...	The bar's or dial's minimum and maximum.
Limits... / Clear limits	Alarm limits (Section 9.4).
Set value...	Only for measurements the A2L file marks as writable (Section 9.5).
Load value from file...	The same measurements: their value from a calibration file (Section 8.3).
<i>Raster names</i> / Not logged	The raster the measurement is logged in (Chapter 9).
Remove from list / Properties...	Removes the row; shows the object's A2L definition.

The choice applies to all selected rows. Right-click a card's top strip to apply it to every row in the card.

8. Calibrating

Calibration values are edited on the cards of a page. Online, a change is written to the ECU when confirmed; offline, it stays in the workspace until saved ([Chapter 14](#)) or downloaded when going online.

8.1. Editing tables

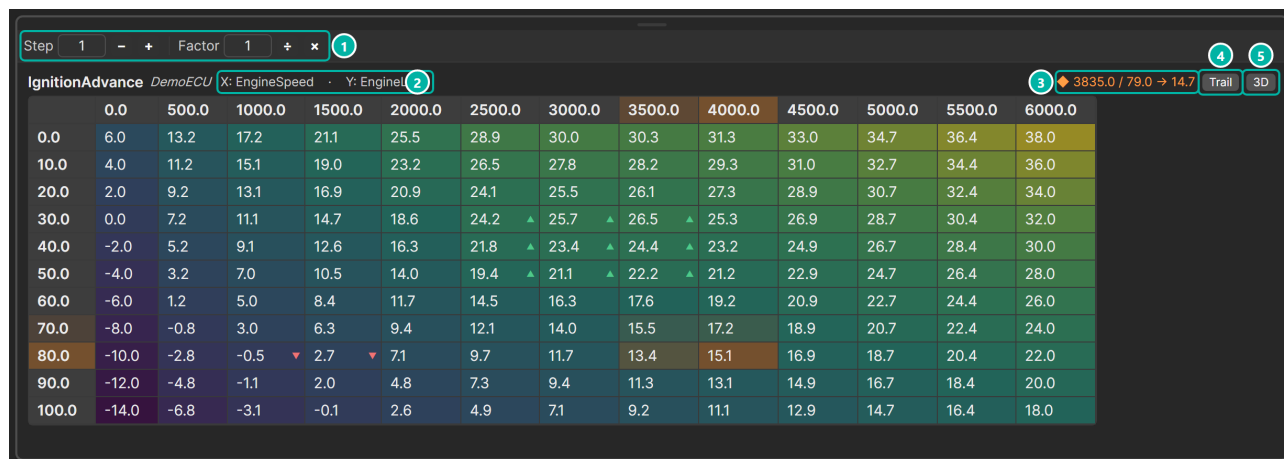


Figure 14. A map card with the live working point and change marks

1. **Step / Factor** — add, subtract, multiply or divide the selected cells.
2. **Axes** — the measurements the map is looked up by.
3. **Working point** — the input values and the resulting map value.
4. **Trail** — also shows where the working point has recently been.
5. **3D** — switches to the 3D view.
 - **Select** cells by clicking or dragging, or with the arrow keys (**Shift** extends).
 - **Step** with the buttons or **+** **-** ***** **/**.
 - **Type** a number and press **Enter**; with several cells selected, all get the value. **Esc** cancels. Start a negative number by double-clicking the cell.

Page Up/**Page Down** move a screen of rows, **Home**/**End** to the row's ends and **Ctrl** + arrow to the table's edge. Chart and 3D views stay editable the same way; drag a 3D surface to rotate it.

8.2. Change marks

A green ▲ or red ▼ marks a value you have raised or lowered compared with the memory image (or, without one, with the ECU at connect). Hover over it to see the original value.

To undo changes, right-click a table: **Revert selected cells** puts the marked values in the selection back, **Revert all changes** every marked value in the table. Online, the reverted values are written to the ECU.

8.3. Loading one value from a file

To take a single object from another calibration — one map from last week's tune, one constant from a colleague's file — without loading the whole file, right-click its card or row and choose **Load value from file....** Pick a **.cdfx**, **.dcm**, **.hex** or **.s19** file; only that object is changed. Online, it is written to the ECU like any edit. If the file does not hold the object, or holds it in another shape, the status bar says so.

8.4. The working point

While recording, the cells the ECU interpolates between are highlighted in orange, and the caption shows the inputs and the result. After a recording, and in a log, the working point follows the plot cursor ([Chapter 9](#)).

To use another input measurement, right-click the card and choose **Working point input**.

8.5. Single values

Press or double-click to edit. Values with named states are chosen from a list; bit fields can be shown as checkboxes (**Display as > Bit checkboxes**).

9. Logging

CalTool reads measurements with XCP DAQ: the ECU sends the values at fixed rates called **rasters** (for example **10ms_Raster**). A measurement is logged when it is assigned to a raster. CAN bus devices use the **CAN raster**. All devices are recorded on one common time axis.



Figure 15. Analysing a recording: a plot across two devices, with readouts, bars and working points at the cursor

1. **At cursor** — after a recording, readouts show the values at the plot cursor.
2. **Bar display** — the value as a bar.
3. **Working point** — where the ECU was operating at that moment.

9.1. Choosing what is logged

Anything dropped onto a page or plot is logged in the device's slowest raster. Change it by right-clicking the measurement ([Section 7.3](#)), or in the **Measurement Rasters** pane:

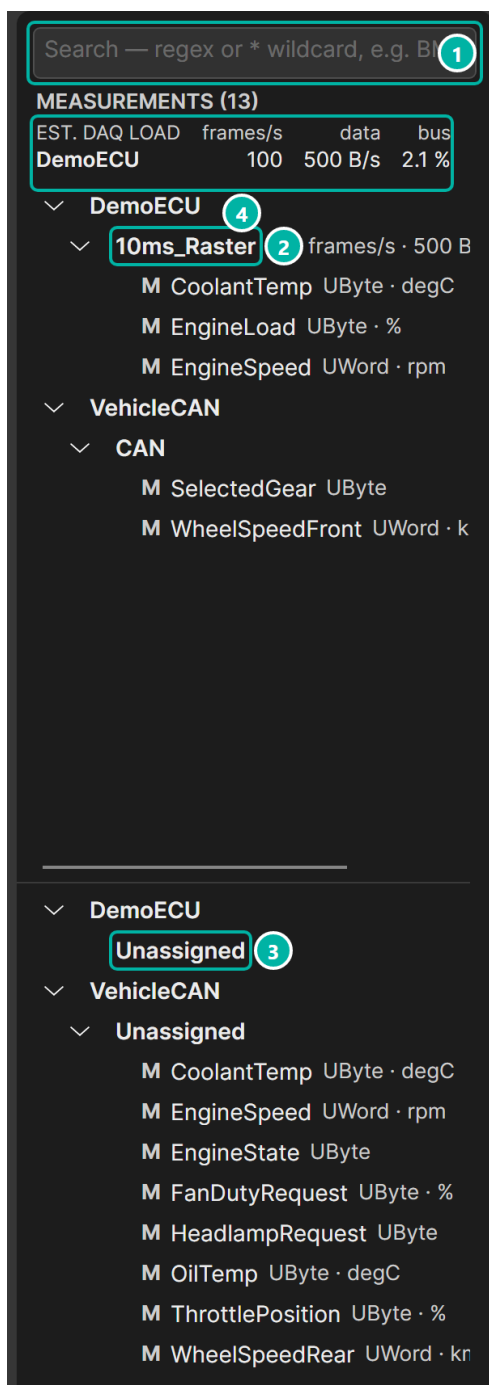


Figure 16. Measurement Rasters pane

1. **Search** — filters the lists.
2. **Raster** — drag measurements between rasters.
3. **Unassigned** — not logged. Select logged measurements and press **Delete** to move them here.
4. **Estimated DAQ load** — frames per second, data rate and CAN bus load per ECU, for the interface and bitrate the device is set to.

Each raster shows its period as declared in the A2L file, or **period unspecified**.

A raster marked holds more measurements than the ECU can send in it; the tooltip names the ones that will not be logged.

9.2. Recording

Click **Record** in the status bar or press . Recording needs a device online, a CAN Trace interface ticked **Record with measurements** (Section 15.4), or a camera (Chapter 12).

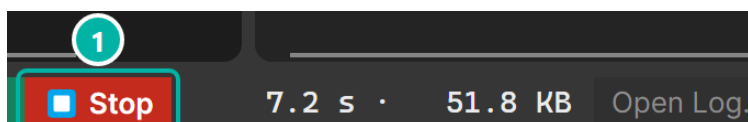


Figure 17. The status bar while recording

1. **Stop** — with the elapsed time and file size.

If the logged signals need more than the bus can carry, CalTool says so before recording: **Cancel**, or **Record Anyway** and accept that the ECU drops samples. While recording, a chip next to the elapsed time warns when samples are missing or the ECU reports an overload; its tooltip says which raster.

Press **M** to mark a moment while recording — a gear change, a noise — and add the comment later (Section 11.8). CalTool also marks it when an ECU stops answering or a CAN bus goes quiet.

After **Stop**, the plot cursor is placed on the last sample. Move it in any plot to see every value, bar and working point at that moment.



Measurements are only read while recording. Going online alone sends no measurement traffic.

9.3. Bars, gauges and highlights

Right-click a measurement row: **Bar display** shows the value as a bar within its A2L range, and **Gauge display** as a dial. **Bar/gauge range...** sets your own range. A dial grows with its card, so a large engine-speed dial can be read from across the bench. **Highlight** colours the row, for example to match its plot line.

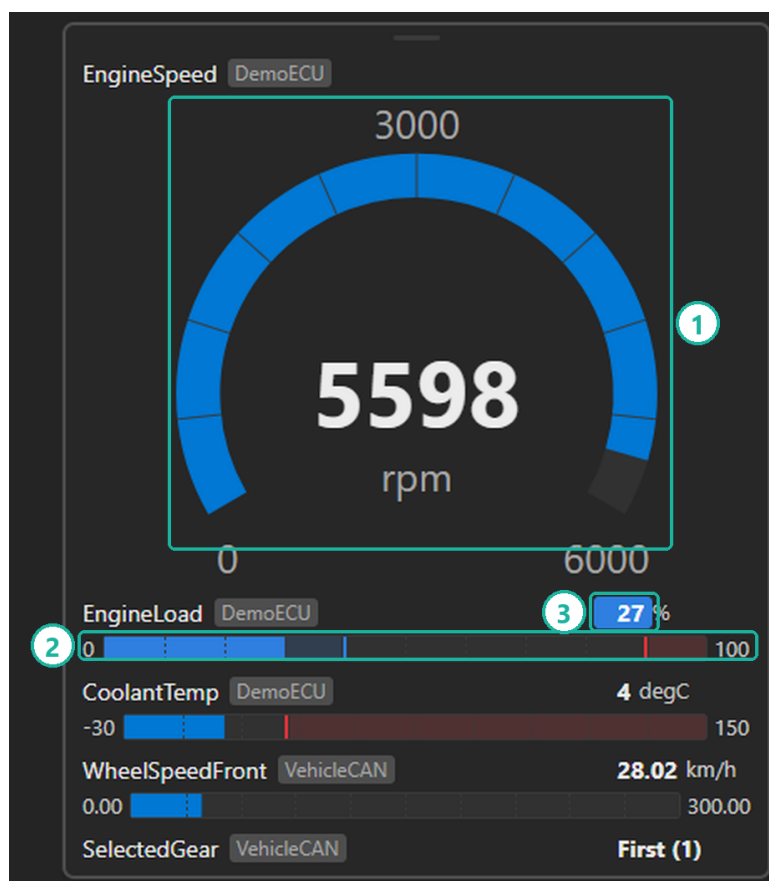


Figure 18. A measurement card with a dial, bars and alarm limits

1. **Gauge display** — the value on a dial.
2. **Bar display** — the value as a bar; the limits are drawn as lines on it.
3. **Limit badge** — red above the high limit, blue below the low one.

9.4. Alarm limits

Limits... sets a **High** and/or **Low** limit for the selected measurements; leave one blank for no limit on that side. When a value goes past a limit, its number is shown on a red (high) or blue (low) badge, and on a bar the area beyond the limit is tinted. **Clear limits** removes them. Limits are saved with the workspace.

9.5. Measurements you can set

Some ECUs let you write certain measurements, typically values kept in RAM or non-volatile memory (the A2L file marks them as writable). Their rows offer **Set value...**: type a number, or choose a named state. Online, the value is written to the ECU at once; offline, it is kept in the workspace and offered for download when you go online. Such values are also loaded from and saved to calibration files with the calibration values.



A value written to ECU RAM is lost when the ECU is switched off, unless the ECU stores it itself.

9.6. Saving a log

Click **Save Log...** (status bar or Logs pane), or stop with **S**. Logs are saved as MDF4 (.mf4), which other measurement tools open directly. CAN traffic recorded with the measurements (Section 15.4) and videos (Chapter 12) are saved in the same file.

Recording_2026-09-23_10-42.mf4

1

Name A. Engineer

Division Calibration

Project Demo vehicle

Measurement object Ignition map, part-load sweep

Comment

Coolant 80-90 degC. Ignition advance +2.5 deg at 2500-3500 rpm, 30-50 % load.

All 6 signals, 1 computed

2

3

4

Include computed channels

Choose signals...

Load Template...

Save Cancel

1. **Description**—stored in the file and shown by other MDF tools, and in the Logs pane (Chapter 10), where it can be edited later.
2. **Include computed channels**—see Appendix B.
3. **Choose signals...**—save only some signals.
4. **Load Template...**—fills in the fields from a template.

Figure 19. Describing the measurement when saving

A log that is already open can be saved again, for example under another name or with new computed channels.

Measurements with named states keep their state names in the log, so they show in CalTool and in other MDF tools even without the A2L file.

9.7. Opening a log

Open Log... (status bar or Logs pane) opens MDF4 (.mf4, .mdf), MATLAB (.mat) and CSV files; the Logs pane also opens them from its table (Chapter 10). The signals appear in the Explorer under **LOG SIGNALS**; cards, plots and working points then show the log at the plot cursor, with no ECU connected. A large log opens in the background, with its progress in the status bar.

- If the ECU names in the log differ from your devices, CalTool asks which is which.
- A log of raw CAN frames from a data logger shows **Pair DBC...** in the Logs pane; choose the bus databases to decode it.
- A damaged log is repaired into a copy; the original is never changed.
- **Time Range...** shows only part of the log (Section 10.2).

9.8. Exporting to MATLAB or CSV

Export Log... in the Logs pane writes the log or recording to **.mat** (one structure per signal, with **time** and **value**) or **.csv** (one column per signal). Values are never resampled or invented.

10. Browsing Logs

The **Logs** pane lists the logs in your folders with the description that was entered when each was saved, opens them, and lets you correct a description afterwards. Everything about log files — opening, saving, exporting and converting — is in this pane.

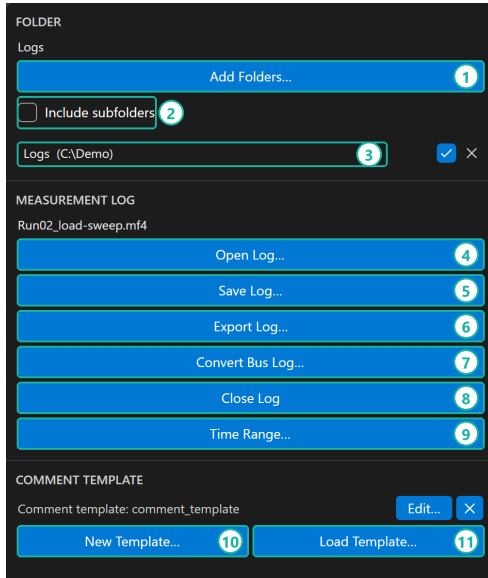


Figure 20. Logs pane

1. **Add Folders...** — adds one or more folders to the list.
2. **Include subfolders** — also lists the logs in their subfolders.
3. **Folder** — tick the folders to list together; click a name to list it alone. The cross forgets a folder.
4. **Open Log...** — [Section 9.7](#)
5. **Save Log...** — [Section 9.6](#)
6. **Export Log...** — [Section 9.8](#)
7. **Convert Bus Log...** — converts a CAN bus log between MDF4, BLF and text ([Section 15.4](#)).
8. **Close Log** — closes the log or discards the stopped recording.
9. **Time Range...** — shows only part of the open log ([Section 10.2](#)).
10. **New Template...** — the description to pre-fill when a log is saved.
11. **Load Template...** — loads a saved template.

A log of raw CAN frames also shows **Pair DBC...**, to choose the databases that decode it ([Section 9.7](#)). Below the comment template, **Markers** lists the open log's markers ([Section 11.8](#)).

10.1. The log table

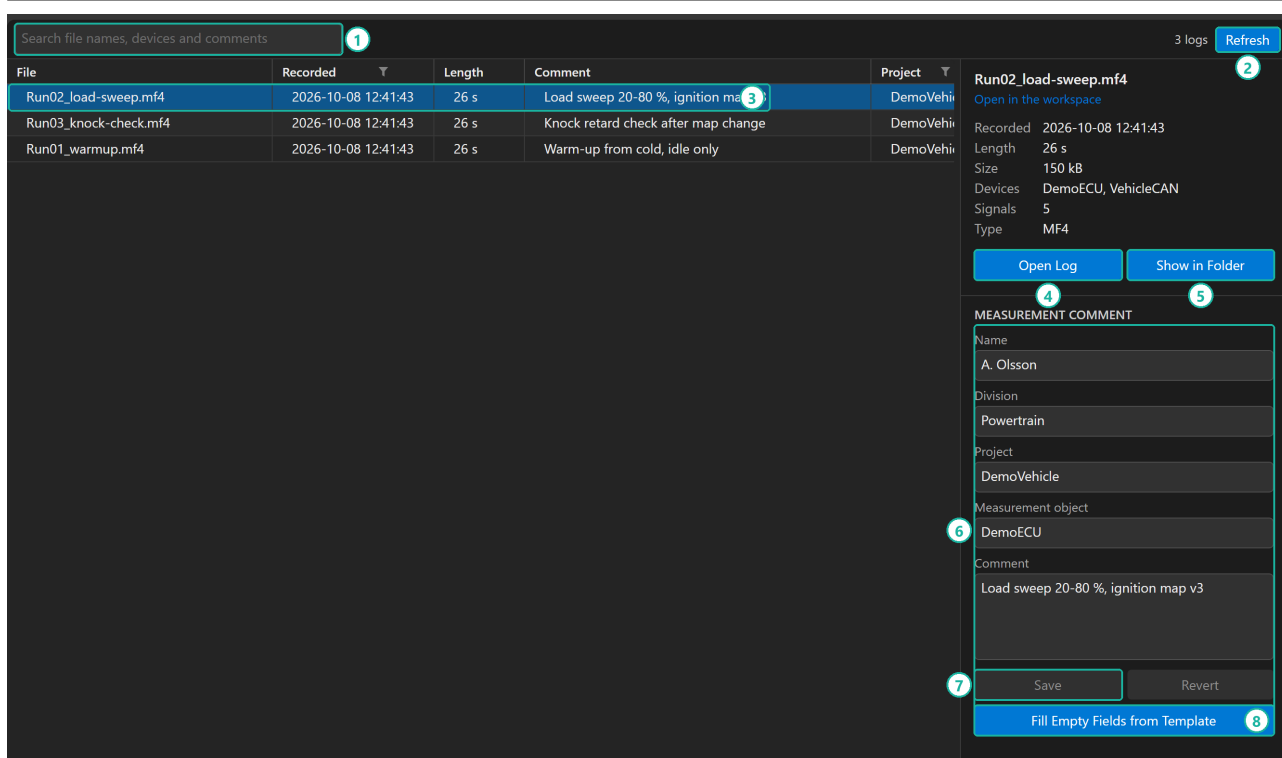


Figure 21. The logs in a folder, with one selected

- Search** — each word must match the file name, folder, devices or description. * is a wildcard.
- Refresh** — reads the folders again. New and changed files are also noticed by themselves.
- Table** — click a column header to sort; the funnel in a header filters by that column's values, or for **Recorded** by date. **Clear filters** appears while a filter is active.
- Open Log** — opens the selected log in the workspace (also double-click or **Enter**).
- Show in Folder** — shows the file in Windows Explorer.
- Measurement comment** — the description stored in the file; edit it here.
- Save** — writes the edited description into the log file (or **Ctrl + S**). Moving to another log saves too. **Revert** discards the edit.
- Fill Empty Fields from Template** — completes a description from the comment template.

Recorded is the time on the PC that made the recording, as the file name says, wherever you read it. **Devices** names the ECUs in the log. Length, devices and signals are read in the background after the table appears, so a folder of hundreds of large logs lists at once.

Descriptions can be edited in MDF4 logs. A BLF, CSV or MAT file has nowhere to store one.

The folders are remembered between sessions. Column widths, order and sorting are saved with the workspace.

10.2. Showing part of a log

Time Range... keeps only the samples between **From** and **To** (in the log's own seconds), and **Offset**

shifts them, for example so an interesting section starts at 0 s. Every plot, readout and export then sees only that part; **Show All** goes back to the whole log. **Save Log...** afterwards writes the shortened, shifted log.

A plot's right-click menu offers **Crop log to the cursors**, which sets the range from the two cursors ([Chapter 11](#)). The range is saved with the workspace.

10.3. Comparing and combining logs

A second log can be opened beside the open one, in one of two ways:

- **Comparison** — the same signals from another run, for example before and after a calibration change. Every signal you plot is drawn a second time from the other log, dashed, on the same axis.
- **Timeline** — another recording of the same drive, for example a CAN bus log next to an ECU recording. Its signals are added to the signal list and can be plotted like the open log's own.

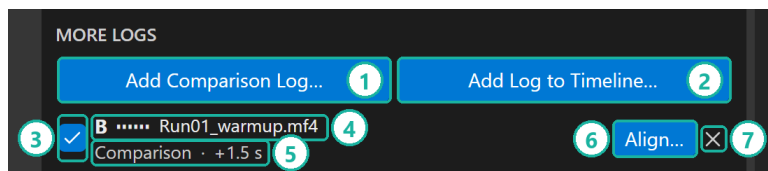


Figure 22. A comparison log beside the open log

1. **Add Comparison Log...** — picks a log to compare with the open one.
2. **Add Log to Timeline...** — picks a log whose signals join the open one.
3. **Checkbox** — shows or hides a comparison log's curves.
4. **Label** — the log's letter (the open log is A), the dash pattern its curves are drawn with, and its file.
5. **Offset** — where this log's time 0 sits on the open log's timeline.
6. **Align...** — lines the log up in time ([Section 10.3.2](#)).
7. The cross takes the log off the timeline.

You can also select a log in the log table and press **Add as Comparison** or **Add to Timeline** under **Open Log**, or right-click it. A log that is open beside the main one is marked in the details with its letter.

When a log was recorded with ECUs that the workspace does not know by name, CalTool asks which ECU each one is, as it does when a log is opened.

Comparison curves

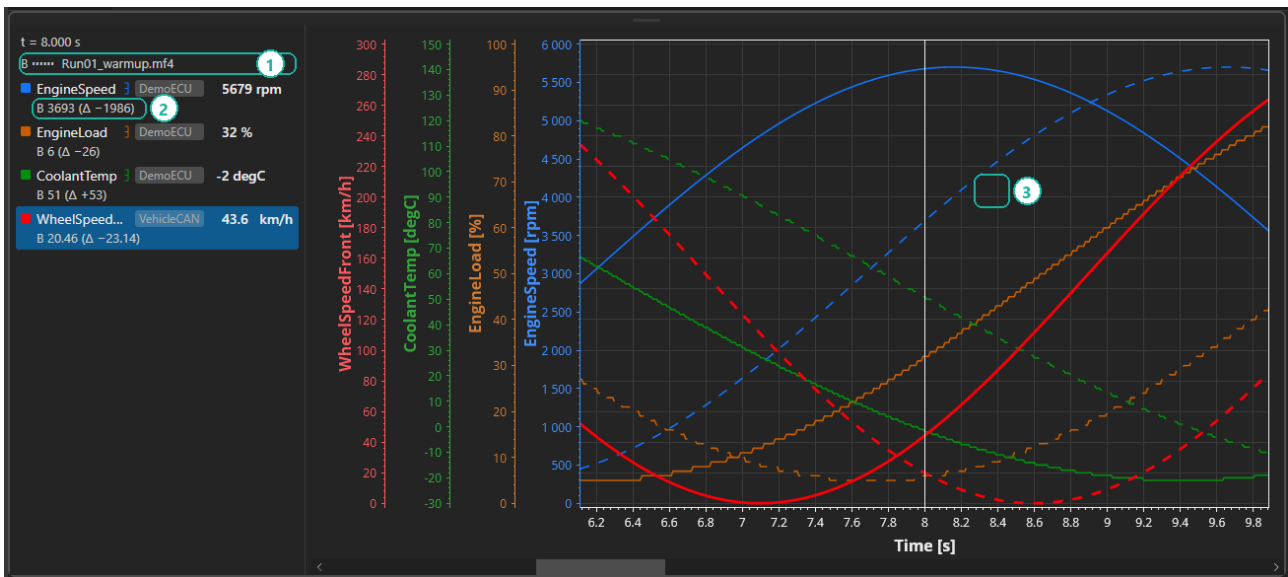


Figure 23. A plot with a comparison log

1. The comparison logs, each with its dash pattern.
2. With the cursor placed: the signal's value in the comparison log, and its difference from the open log's value.
3. The comparison curve: the same colour as the signal, dashed.

Nothing has to be dragged from the comparison log: signals you plot later get their comparison curve too. A signal that is not in the comparison log simply has none. Several comparison logs can be open at once; each gets its own dash pattern.

Lining logs up in time

Two runs rarely start at the same moment, so a log beside the open one can be moved in time. **Align...** opens a dialog with the offset in seconds. Type it, or use **Align at cursors**:

1. Place the cursor on an event in the open log's curve, for example a pedal step.
2. **Shift**+click the same event in the dashed curve to place the compare cursor there.
3. Press **Align...**, then **Align at cursors**. The offset field shows the result.
4. Press **OK**. The two events are now on top of each other.

A log added to the timeline also starts at offset 0; set its offset the same way if the two recordings did not start together.

The logs beside the open one, with their offsets, are saved with the workspace. Closing the open log, or opening another one, closes them too.

11. Plotting

Plots are cards. Create one by right-clicking a measurement list and choosing **Convert to plot** (or **Convert all to plot**), then drag more signals onto it from the Explorer. **Convert to list** in the plot's right-click menu goes back.

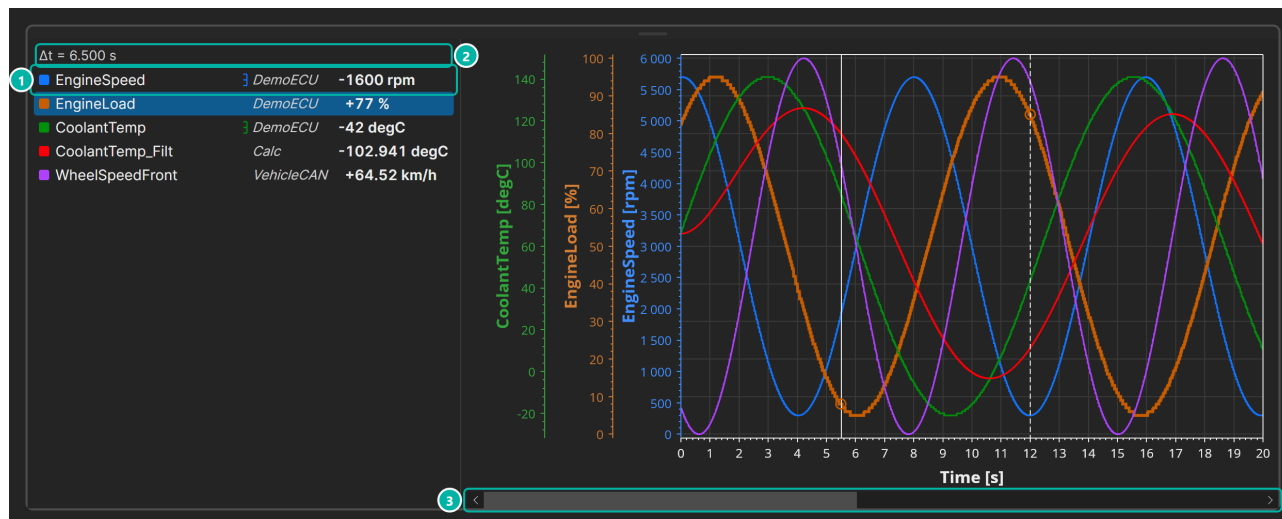


Figure 24. A plot after a recording, with two cursors and two pinned Y axes

1. **Legend** — click a row to select the signal and show its Y axis. Each row shows the value at the cursor; with two cursors, also the difference between them.
2. **Time readout** — the cursor time, or the times of both cursors and Δt between them.
3. **Time scroll bar** — pans in time.

Action	Result
Click or drag in the chart	Places the cursor (A)
Shift + click in the chart	Places the compare cursor (B)
Alt + drag in the chart	Moves both cursors together
Right-click in the chart	Cursors, subplots, grid, X axis, statistics, picture, convert to list
Right-drag across the chart	Zooms to that time range; dragged up or down, zooms the selected signal's Y axis
Mouse wheel over an axis	Zooms that axis
Double-click an axis	Type the minimum and maximum
Arrow keys	Pan
Ctrl + ← / Ctrl + →	Steps the cursor one sample of the selected signal; add Shift for the compare cursor
Drag a legend row	Reorder, or move to another plot or card
Ctrl / Shift + click, Ctrl + A in the legend	Select several signals
Delete / Space	Remove / hide the selected signals

Right-click a legend row to hide the line, **Pin Y axis** (keep it visible), change **Line color**, raster or

display format, show **Dots only** (the samples without a line), type a **Y axis range...**, reset the Y axis, or remove the signal. With several rows selected, the entry applies to all of them. Zoom, colours and pinned axes are saved in the workspace.

11.1. Cursors

The cursor (A) marks one moment in every plot, readout and working point. The compare cursor (B) measures against it: the legend shows **A**, **B** and Δt above the signals, and each row the difference between B and A. Place B with **Shift**+click, or from the chart's right-click menu, which also offers **Zoom to cursors**, **Swap cursors A and B** and **Remove compare cursor**. Only a press on B's line moves B, so you can work with A without disturbing it.

When a cursor is outside the visible time range, an arrow at the edge of the chart points to it: click it to scroll there, or drag it to pull the cursor into view.

11.2. A plot as a picture

Save as picture... in the chart's right-click menu saves the plot as a PNG, exactly as shown — subplots, legend with its values, axes and cursors — at twice the screen resolution. **Copy as picture** puts it on the clipboard, to paste into a report or e-mail.

11.3. Gaps in a recording

When an ECU stops answering during a recording ([Section 5.3.1](#)), its curves break at that point and its last value is drawn dashed until it answers again. Markers show where the connection was lost and restored.

11.4. Time across plots

With **Sync time** in the status bar on (the default), every plot shows the same time range: zooming or panning one moves them all. Off, each plot keeps its own range. A plot whose range does not contain the cursor moves to it when the cursor is placed elsewhere.

11.5. Subplots

Add subplot in the chart's right-click menu puts another plot under this one, in the same card, sharing the time axis. Each subplot has its own signals, legend and Y axes; drag the line between them to change their heights. **Remove subplot** takes one away with its signals.

11.6. A signal as the X axis

Use as X axis on a legend row draws every other signal against that signal's values instead of time — for example torque against engine speed. The row is marked **X**, and the chart's bottom axis shows the signal's range. The cursor still marks a moment: click a point to put the cursor at its time in every plot. **X axis > Time** in the chart's menu goes back.

11.7. Between the cursors

With the compare cursor placed, the chart's menu **Between cursors** adds **Min**, **Max** and **Average** of each signal between the two cursors to its legend row, next to the difference (**Delta**). The choice applies to every plot and is saved in the workspace.

11.8. Markers

A marker is a note at a moment in a log: a name, an optional comment, and either one time or a span. Markers are drawn on every plot as a dashed line with a flag carrying the name; hold the pointer over a flag to read the comment. Markers that other tools wrote into an MDF4 log—for example CANape's when it lost an ECU—are shown too.

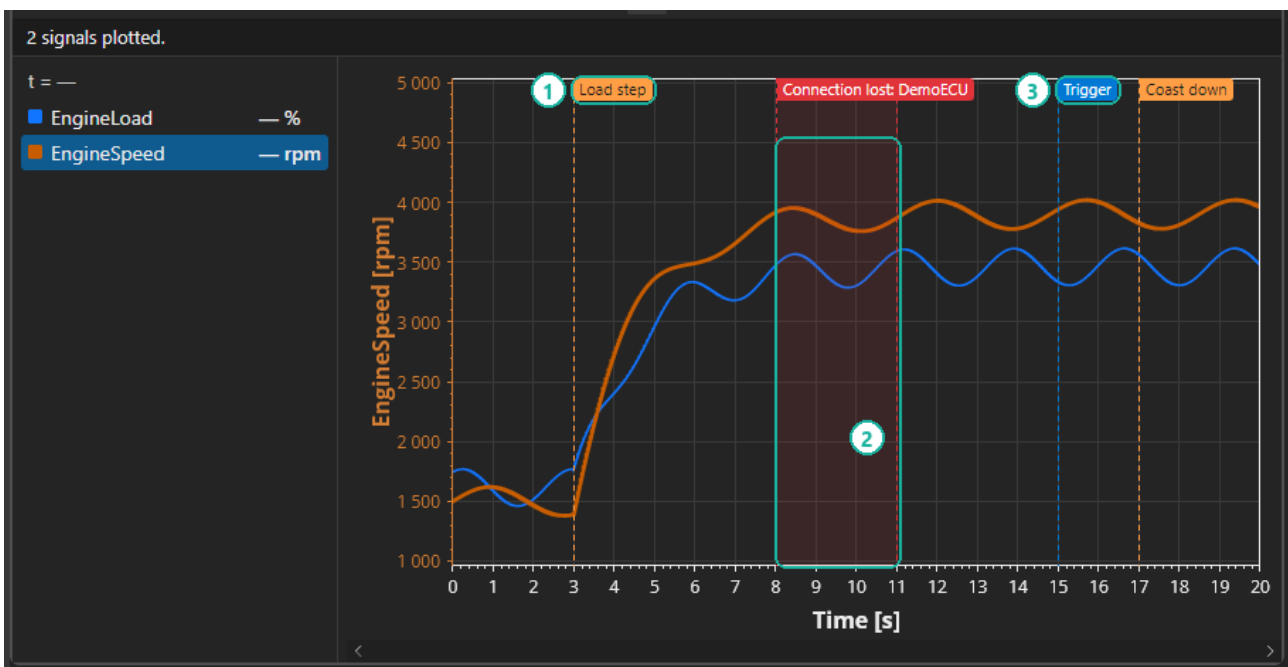


Figure 25. Markers in a plot

1. **Marker** — orange: a note made by you or another user.
2. **Interruption** — red: the measurement was interrupted, for example an ECU stopped answering. A span is shaded from start to end.
3. **Trigger** — blue: a trigger or other event written by a tool.

On a flag: click to put the cursor there, drag to move the marker, double-click to edit it, or right-click to edit, go to or delete it.

To add a marker:

- Press **M**. While recording, the marker is placed at that moment; otherwise at the cursor. It is named **Marker 1**, **Marker 2**, ... — add the comment later.
- Right-click in a plot and choose **Add marker here...**, to type the name and comment straight away. Fill in **End** to mark a span instead of a single moment.

While recording, CalTool marks it when an ECU stops answering or a CAN bus goes quiet, so you can find the gap in the log later.

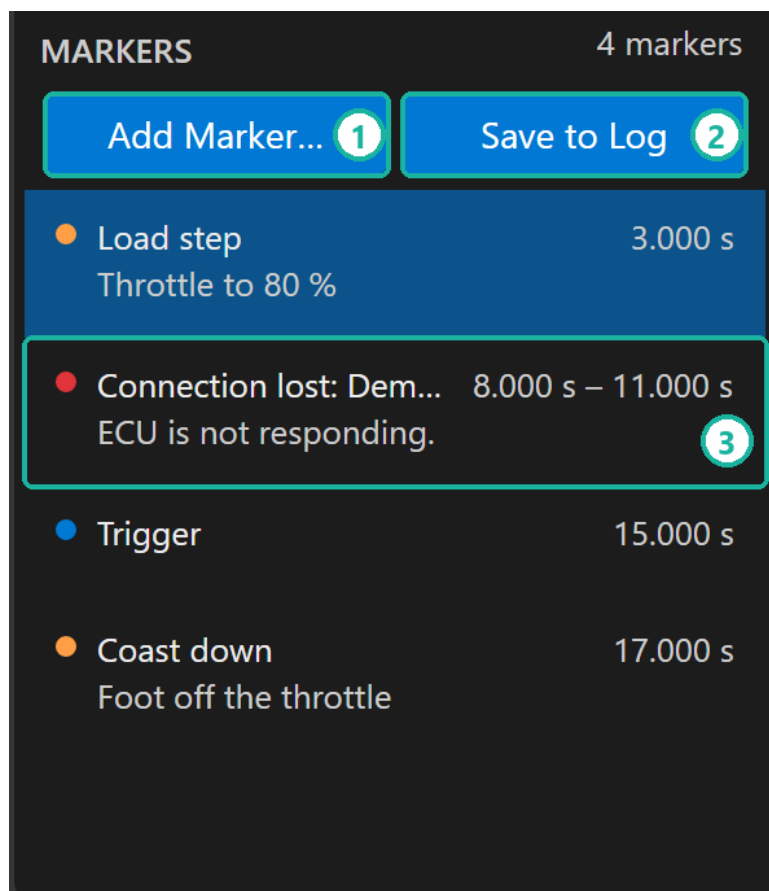


Figure 26. The markers in the Logs pane

1. **Add Marker...** — adds a marker at the cursor (or now, while recording) and asks for its name and comment.
2. **Save to Log** — writes changed markers into the open log's file.
3. **Marker list** — click a marker to put the cursor on it, double-click to edit it, or press **Delete** to remove it.

The markers of a recording are saved with it by **Save Log...** Markers added to or changed in an opened log are written into that file with **Save to Log**; the recorded data is not touched. If you close the log or open another with unsaved markers, CalTool asks whether to save them. Markers can be saved into MDF4 logs only.

12. Video and maps

A video recorded together with a log shows what the vehicle was doing at each moment. CalTool plays the video on a workspace page, locked to the log: the **plot cursor** decides which frame is shown, and playing the video moves the cursor, so plots, readouts and working points replay with it. A GoPro video also brings the camera's GPS, accelerometer and gyro as signals, and its GPS track can be shown on a map.

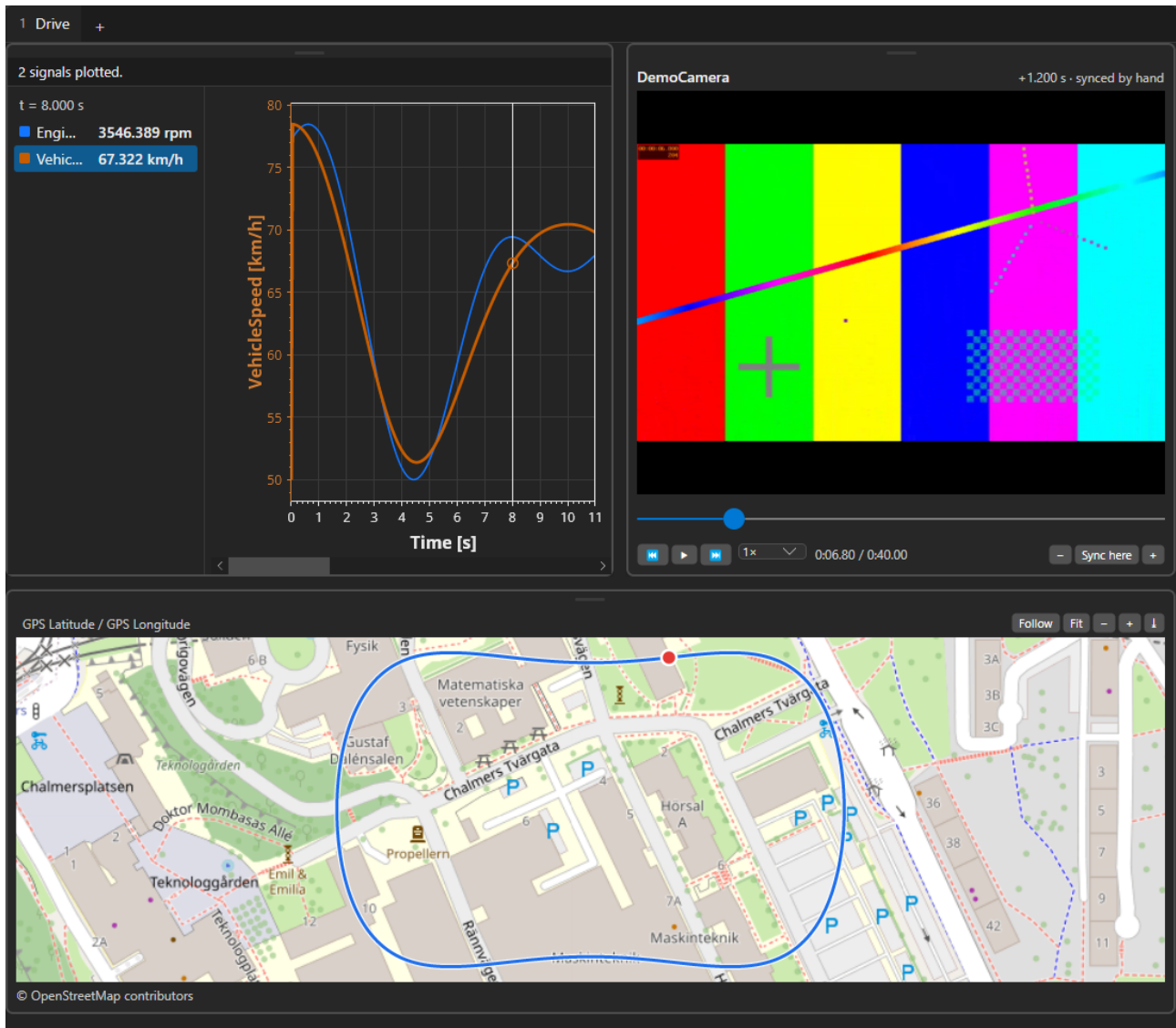


Figure 27. A log, its video and its GPS track at the same moment

12.1. Adding a video

Videos are managed in the **VIDEO** section of the Files pane.

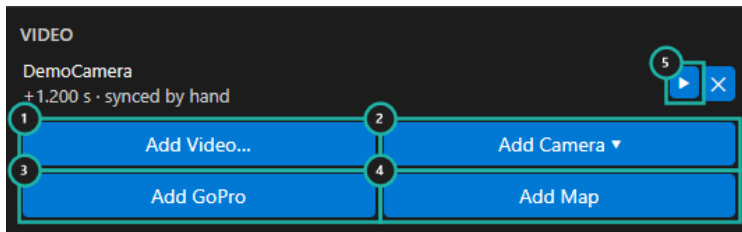


Figure 28. The VIDEO section

1. **Add Video...**—a video recorded elsewhere, for example a GoPro's files.
2. **Add Camera**—a camera on this PC, recorded together with **Record** (Section 12.4).
3. **Add GoPro**—a GoPro controlled by CalTool (Section 12.5).
4. **Add Map**—a map of a GPS track (Section 12.7).
5. —shows the video on the current page. removes it (the file is not deleted).

Under each video, the line shows where it sits against the log and how that was decided, for example **+1.200 s · GPS time**. A positive value means the video started after the log.

A GoPro recording longer than about 4 GB is split into several files. Pick any one of them; the others are added with it.

With no log open, adding a GoPro video opens its GPS, accelerometer and gyro as the log. A GoPro video can also be opened directly with **Open Log...**

12.2. The video card



Figure 29. A video card

1. **Offset**—where the video sits against the log.
2. **Play** (Space)—plays the video and moves the plot cursor with it. step one frame. The speed box plays slower or faster.
3. **Scrub bar**—moves the plot cursor through the video.
4. **Sync here**—the frame on screen belongs at the plot cursor. See Section 12.3.
5. **-/+**—moves the video one frame earlier or later against the log.

When the plot cursor is before or after the video, the picture is dimmed and says so.

A GoPro video with GPS has a **GPS** button, which opens a map of its track.

12.3. Lining a video up with the log

CalTool places a video automatically when it can:

- **GPS time** — a GoPro with a GPS fix. Exact.
- **Recorded with the log** — a camera recorded by CalTool. Exact.
- **Record command** — a GoPro controlled by CalTool without a GPS fix. Within a few tenths of a second.
- **Camera clock** — the time stamped in the file by the camera. Often seconds off; check it.

To line a video up by hand:

1. Pause the video on an event you can also see in a plot, for example a brake light coming on or a gear change.
2. Put the plot cursor on the same event in the plot.
3. Click **Sync here**.

Fine-tune with $-/+$, or type the offset with **Offset...** in the card's right-click menu.

The sync is saved in the workspace and, when the log is saved, in the log file.

12.4. Recording video with a log

Add Camera lists the cameras on this PC. The card shows the live picture, so the camera can be aimed before recording.

Click **Record** as usual: the camera is recorded together with the measurements, already in sync. After **Stop**, the card plays the recorded video. **Live** shows the live picture again.

12.5. Recording with a GoPro

CalTool can control a GoPro HERO9 or later:

1. Connect the GoPro by USB and switch it on. In the camera's settings, set the USB connection to **GoPro Connect**. Alternatively, connect the PC to the GoPro's own Wi-Fi.
2. Click **Add GoPro** in the VIDEO section. CalTool finds the camera and sets its clock to the PC's. The status line shows the battery and whether the GPS has a fix.
3. The card shows the camera's preview, to aim it.
4. Click **Record**: the GoPro starts recording. Click **Stop**: the GoPro stops.

After **Stop**, CalTool fetches the recording from the camera:

- The GPS, accelerometer and gyro arrive within seconds, as signals under the device **GoPro**.
- Then a low-resolution copy of the video, which the card can play at once.
- Then the full video.

Some GoPro models stop the preview while recording; the card then shows the camera's status instead.

12.6. GoPro signals

A GoPro video adds these signals under the device **GoPro**:

Signal	Unit	Description
Accel X, Accel Y, Accel Z	m/s ²	Acceleration in the camera's axes (Z includes gravity).
Gyro X, Gyro Y, Gyro Z	deg/s	Rotation rate in the camera's axes.
GPS Latitude, GPS Longitude	deg	Position. Only samples with a GPS fix.
GPS Altitude	m	Height above sea level.
GPS Speed, GPS Speed 3D	km/h	Ground speed, and speed including height changes.
GPS Fix		2 = 2D fix, 3 = 3D fix.
GPS DOP		Precision of the fix; lower is better.

They plot, read out and save like any other signal, and move with the video when it is synced.

12.7. The map card

The map card shows a GPS track over a map, with a marker at the plot cursor. It works with a GoPro's GPS or with any latitude/longitude pair in the log, for example a GPS on the vehicle's CAN bus.

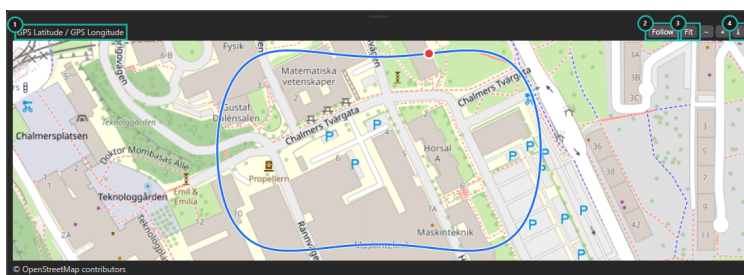


Figure 30. A map card

1. **Signals** — the latitude and longitude signals the track is drawn from. Click to change them.
2. **Follow** — keeps the cursor's position in view.
3. **Fit** — shows the whole track.
4.  — downloads an area for offline use (Section 12.8).

Drag to move the map and use the mouse wheel to zoom. Click on the track to move the plot cursor to the moment the vehicle was there.

12.8. Maps without internet

Every part of the map you have looked at is kept on the PC and is available without internet afterwards.

To prepare an area in advance, for example a test track:

1. Click  on the map card and drag a rectangle over the area.
2. Choose the zoom levels. The dialog shows how many map tiles that is and roughly how much space it takes.
3. Click **Download**.



OpenStreetMap, the default map, does not allow downloading areas. To download areas, use a map provider that allows it: right-click the map card, choose **Map source...**, and enter the provider's tile address (usually including your API key), its attribution text, and tick that the provider allows downloading.

12.9. Saving a log with its video

Save Log... saves the videos with the log, in the standard MDF4 way that other measurement tools understand:

- A video recorded by CalTool (camera or GoPro) is moved next to the log file and named after it, for example **Run12_DemoCamera.mp4**.
- A video you added stays where it is.

The log refers to the video files, it does not contain them. Keep the videos next to the log when copying or sending it.

Opening the log again (**Open Log...**) adds its videos back, in sync.

13. Working with an AI assistant

Altech CalTool can be used together with an AI assistant such as Claude or ChatGPT. The assistant connects to the CalTool you have open and works in that session: it reads the signals, logs and calibration data you have loaded, and—as far as you allow—changes the workspace and the calibration. Everything it does happens in your window, where you see it, and every step is listed in CalTool.

What an assistant can do for you:

- **Analyse a log:** find where a signal went out of range, when a state changed, how two signals relate, what is unusual in a recording—and mark the findings on the timeline with a comment, so you can look at them in the plots.
- **Build computed channels:** describe the quantity you want (a filtered signal, a rate, a power estimate, an edge counter) and the assistant writes and tests the script, then adds the channel for you to plot and record ([Appendix B](#)).
- **Set up pages:** open plots and readouts for the signals a question needs, place the cursor at an event, and check the picture.
- **Explain calibration objects:** what a table does, how its axes relate to the measurements around it, where its values look unusual.
- **Calibrate:** change values, curves and maps—offline into the workspace, or on an online ECU after you have confirmed the change.

The assistant does not replace the calibration engineer. It works from the names, units and values in the A2L and the logs, and it is only as right as its reasoning; read its conclusions the way you would read a colleague's.

13.1. Switching AI access on

Open **Settings** (the gear at the bottom of the activity bar) > **AI access** > **Configure....**

☒ **Let an AI assistant use this CalTool** 1

Listening on `http://127.0.0.1:38777/mcp`

What it may do

☒ Read signals, logs, calibration values and the workspace

☒ Change the workspace: plots, cards, markers, computed channels, logs, recordings 2

☒ Change calibration values on the desktop and in files

☐ Write to an online ECU (each batch is confirmed in a dialog)

Connect

Client **Claude Code (terminal)** 3 Port `38777` **New token** 4

1. Open PowerShell (Start menu → type PowerShell).
2. If the `claude` command is not installed yet, run: `irm https://claude.ai/install.ps1 | iex`
3. Press Copy, paste the text below into PowerShell and press Enter. This is done once.
4. Start Claude Code (type `claude`) with CalTool running and AI access on, and ask it to use CalTool.

```
claude mcp add --transport http caltool http://127.0.0.1:38777/  
mcp --header "Authorization: Bearer k7Rw2QpXv9TzLm4HbNc3YdFg"
```

Copy

Activity Clear

13:47:17 **add_marker** marker 'Coolant warm' at 12.4 s

13:47:17 **add_computed_channel** added computed channel Calc/CoolantRate 5

Close

Figure 31. The AI access dialog

1. **Let an AI assistant use this CalTool**—the switch. Off, nothing listens and no assistant can connect.
2. **What it may do**—reading is always allowed while the switch is on; each of the three kinds of change is allowed only when ticked (Section 13.2).
3. **Client**—which AI tool you connect; the text box and the note below say what to copy and where to put it (Section 13.3).
4. **New token**—replaces the secret the assistant must present. Every connected client then has to be set up again.
5. **Activity**—every request the assistant has made in this session, newest first, with what it did or why it was refused.

While AI access is on, an AI chip at the right end of the status bar lights up on every request, so an assistant never works unnoticed. Clicking the chip opens this dialog.

The connection is only accepted from this computer, and only with the token shown in the dialog. The settings are kept per computer, not in the workspace.

13.2. What the assistant may do

Permission	What it allows
<i>(always, while on)</i>	Reading: the devices, signals and their definitions, live values, the loaded log or recording, calibration values, markers, pages. Analysing the data: statistics, finding events, trying a script, rendering a plot as a picture.
Change the workspace	Adding, changing and removing computed channels and markers; opening plots, readouts and calibration cards; pages; moving the cursor; assigning rasters; opening and closing logs; starting, stopping and saving recordings; saving the workspace. Everything here is visible on screen and can be undone the ordinary way.
Change calibration values on the desktop and in files	Changing values, curves, maps and arrays in the workspace while the device is offline (the cards show the change marks), and saving calibration files. Nothing reaches an ECU.
Write to an online ECU	The same changes while the device is online, where they are written to the ECU — and going online or offline. Every batch of changes is first shown to you in a dialog, old value against new, with the reason the assistant gave; nothing is written until you confirm.

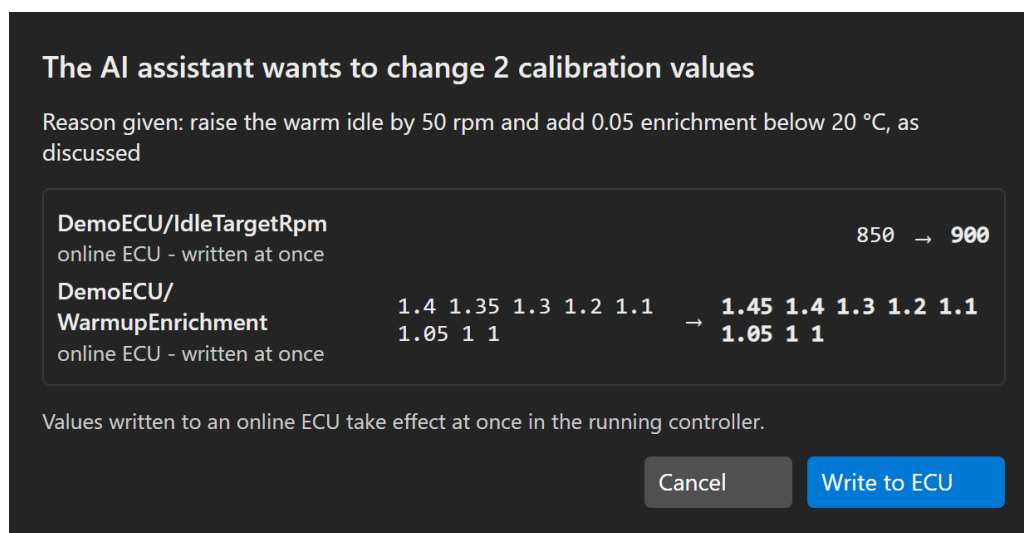


Figure 32. Confirming a change to an online ECU

Values are checked against the A2L before they are shown: a value outside an object's limits, a curve with the wrong number of points, or an object on a read-only calibration page is refused and the assistant is told why. A permission can be changed at any time, also while an assistant is connected; its next request follows the new setting.



Start with reading and workspace changes. Allow calibration changes when you

have seen how the assistant works with your data, and allow writes to an online ECU only for a session where you are sitting at the bench and want them.

13.3. Connecting an AI tool

In the dialog, pick the **Client** you use. The text box fills with what to copy, and the note above it says where it goes. The steps for each:

Claude Code (terminal)

1. Open PowerShell.
2. If the `claude` command is not installed yet, run `irm https://claude.ai/install.ps1 | iex.`
3. Press **Copy**, paste the line into PowerShell and press Enter. This is done once.
4. Start Claude Code (type `claude`) with CalTool running, and ask it to use CalTool.

Claude Code in VS Code

1. In VS Code, create a file named `.mcp.json` in the folder you have open.
2. Press **Copy**, paste the text into that file and save it.
3. Start a new Claude Code chat. It asks whether to use the project's `caltool` server; answer yes.

Claude Desktop

1. Press **Copy**.
2. In Claude Desktop open the menu (≡ at the top left) > **File** > **Settings...** > **Developer** > **Edit Config**. A folder opens with the file `claude_desktop_config.json`.
3. Open that file in Notepad, replace its contents with the copied text, and save. If the file already lists other servers, add only the `caltool` part inside `mcpServers`.
4. Quit Claude Desktop completely—right-click its icon in the system tray and choose **Quit**—and start it again. The CalTool tools then appear under the chat box's tools button.

Claude Desktop starts a small bridge program that CalTool installs; nothing else needs to be installed.

ChatGPT

ChatGPT runs on OpenAI's servers, so it needs a public web address that reaches your computer.

1. Install a tunnel program, for example `cloudflared` or `ngrok`.
2. Start it in PowerShell: `cloudflared tunnel --url http://127.0.0.1:38777`. It prints a web address beginning with `https`.
3. In ChatGPT: **Settings** > **Connectors** > **Create**. Name it CalTool; as the URL enter the tunnel's address followed by the path shown in the dialog (from `/mcp/` on). Authentication: none—the secret is part of the URL.

Custom connectors need developer mode in ChatGPT's settings. Keep the tunnel window open while you work, and stop it afterwards.

Cursor, VS Code, Gemini CLI, other

The text box holds the standard MCP server entry; paste it into the tool's MCP configuration and restart the tool.

CalTool speaks the Model Context Protocol (MCP), the open standard most AI tools use to connect to applications, so any assistant that supports MCP can be connected. If your company uses another AI platform, or runs its own model, contact Altech.

In every case CalTool must be running with AI access switched on when the assistant connects.

13.4. Working with the assistant

Ask in plain language, as you would ask a colleague who has CalTool open. The assistant finds signals by name; where two devices declare the same name, say which device, or use **Device/Name**. Some examples:

- *"What is loaded in CalTool right now?"*
- *"In the open log, where does coolant temperature pass 80 °C, and what were engine speed and load at that moment? Put a marker there."*
- *"Make a computed channel with the rate of change of coolant temperature in °C per minute, and plot it with the temperature."*
- *"Explain the warm-up enrichment table and tell me where its values look unusual."*
- *"Compare engine speed between the two cursors and the ten seconds before them."*
- *"Raise the idle target by 50 rpm."*—with calibration changes allowed; on an online ECU the confirmation dialog appears.

What to expect:

- The assistant reads before it changes anything, and tells you what it did. Check the **Activity** list in the AI access dialog when in doubt.
- Live values update only while a recording captures, as they do on screen.
- A picture of a plot can only be rendered for a plot that is visible; the assistant will ask you to switch pages, or open a plot of its own.
- Changing a curve or map replaces the whole table; the assistant is instructed to read it first and change only the cells it means to, and the confirmation dialog shows what it proposes.
- If a request is refused, the reason names the permission to tick. Grant it in the dialog rather than working around it.

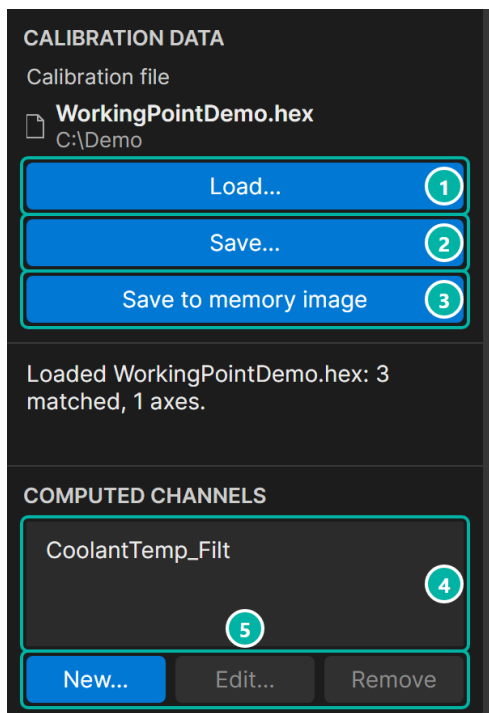
13.5. If the assistant cannot reach CalTool

- The AI access switch is off, or CalTool is not running — the assistant reports it.
- The token was regenerated — set the client up again with the new text, and restart it.
- Another program uses the port — the status line in the dialog says so; change **Port** and set the client up again.
- Claude Desktop shows no CalTool tools — quit it from the system tray (closing the window is not enough) and start it again; then check **Settings > Developer** for the server's state.

14. Calibration Data

Calibration values can be loaded from and saved to **.cdfx** files (ASAM, exchangeable with other tools), **.dcm** files (the DAMOS/ETAS text format) and memory images (**.hex**, **.s19**, **.s28**, **.s37**, **.srec**).

14.1. Loading and saving



1. **Load...** — reads a **.cdfx** or **.dcm** file, or a memory image. Online, the values go straight to the ECU.
2. **Save...** — saves to a **.cdfx** or **.dcm** file, or into a memory image.
3. **Save to memory image** — writes into the loaded memory image, overwriting it.
4. **COMPUTED CHANNELS** — see [Appendix B](#).
5. **New... / Edit... / Remove**

Figure 33. Files pane: calibration data and computed channels

With several ECUs, a **Device** selector chooses which one the files belong to.

Saving into a memory image changes only the calibration values in an existing image (the last one loaded, or the device's memory image from ECU Setup); everything else in the file stays as it was. The Flash Firmware pane then offers that image for flashing ([Chapter 16](#)).

A curve or map whose breakpoints CalTool has not read — no memory image loaded and the ECU not online — is saved to **.cdfx** without its axis, and left out of a **.dcm** file (that format cannot hold a table without its axis). The save names the objects concerned.

To take only one object from a file, see [Section 8.3](#).

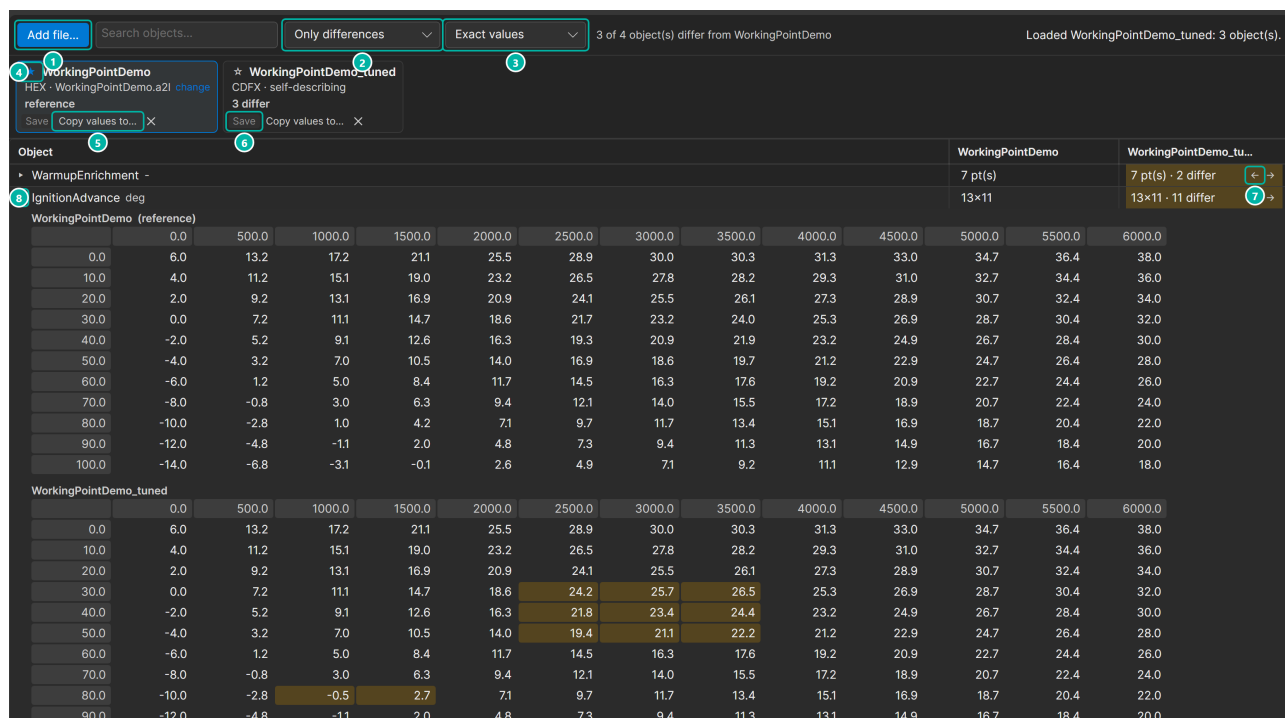


Save to memory image overwrites the file. Use **Save...** with a new name to keep the original.

14.2. Comparing calibration data

The **Compare Calibration Data** pane shows several calibration files side by side. It works on the

files only, not on the workspace.



1. **Add file...** — a .cdfx or .dcm file, or a memory image (with its A2L file).
2. **Filter** — for example **Only differences**.
3. **Precision** — **As displayed** ignores differences too small to show.
4. **Star** — the reference the other files are compared with.
5. **Copy values to...** — copies all values into another file.
6. **Save** — writes the changes back to that file.
7. **← / →** — copies one object to or from the reference.
8. **Expand** — shows a curve or map with the differing cells highlighted.

Amber is a changed value, green an object only this file has, red one it is missing.

15. CAN Trace

The **CAN Trace** pane shows, decodes, logs, sends and replays raw CAN traffic on one or more buses. To log bus signals together with ECU measurements, add the DBC file as a device instead ([Section 5.5](#)).

15.1. Watching a bus

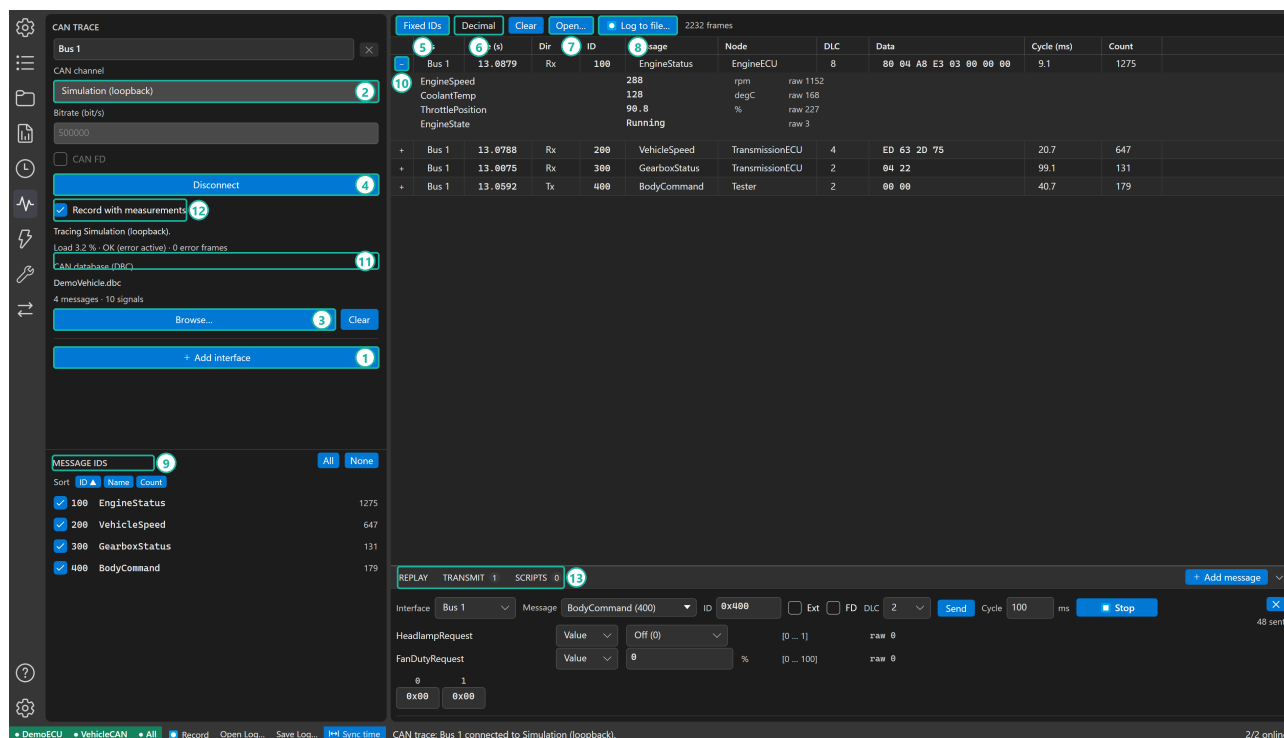


Figure 35. CAN Trace with a DBC loaded and one message expanded

1. **+ Add interface** — trace several buses at once.
2. **CAN channel** — the interface to listen on. Tick **CAN FD** for a CAN FD bus.
3. **Browse...** — the DBC file that names messages and signals.
4. **Connect bus / Disconnect**
5. **Fixed IDs** — one row per message, updated in place, instead of a scrolling trace.
6. **Decimal** — decimal instead of hexadecimal.
7. **Open...** — loads a **.blf**, **.mf4** or **.txt** trace file.
8. **Log to file...** — records the traffic to **.blf**, **.mf4** or **.txt**. Click again to stop.
9. **MESSAGE IDS** — untick an identifier to hide it.
10. **+ / -** — shows a message's decoded signals.
11. **Bus status** — load, controller state and error frames.
12. **Record with measurements** — **Record** in the status bar then connects this interface and records its traffic into the log, next to the measurements ([Section 15.4](#)).
13. **REPLAY / TRANSMIT / SCRIPTS** — the tabs of the panel below the trace. Click the open tab to fold

the panel away; drag its top edge to change its height.

A CAN FD frame shows its length and **FD** in the DLC column. An interface that another pane or program already holds shows that session's bitrate, read-only.

15.2. Sending messages



Figure 36. Sending a message defined in the DBC file

1. **+ Add message**
2. **Interface** — where it is sent.
3. **Message** — from the DBC file, or **raw frame**. The list has a search box: a name, part of a name, or an identifier.
4. **ID** — tick **Ext** for 29-bit.
5. **DLC** — the data length. **FD** for a CAN FD frame (the interface must be ticked **CAN FD**).
6. **Data bytes**
7. **Signal value** — physical values or named states, converted into the bytes.
8. **Send** — once.
9. **Start cyclic / Stop** — repeatedly.
10. **Cycle** — the cycle time in ms.

Numbers are decimal; write **0x** for hexadecimal.

A signal can also be filled in automatically on every send — a rolling counter or a CRC/E2E checksum. For that, and for scripts that react to received messages, see [Appendix C](#).



Sent messages go onto the real bus and can make ECUs react.

15.3. Replaying a log

The **REPLAY** tab sends a recorded CAN log back onto the bus, with its original timing.

1. Click **Load log...** and choose a **.blf**, **.mf4** or **.txt** log.
2. For each channel in the file, choose the interface to send it on, or **Not sent**.
3. Untick the identifiers that must not be sent. **Include Tx frames** also sends the frames the logger

itself transmitted.

4. Set **Speed** (1 = real time) and **Loop** if the log should repeat.
5. Click **Start**. **Pause** holds the replay; **Stop** ends it.

The sent frames appear in the trace as **Tx**. The replay setup is saved with the workspace, but never started by opening it.

15.4. CAN logs and the measurement log

CAN traffic can be part of a measurement log: tick **Record with measurements** on the interface cards, then **Record** and **Stop** as usual. The frames are saved in the **.mf4** file in the standard MDF4 bus-logging layout, which other tools decode with a DBC file. When saving, the dialog shows how many frames are included; untick it to leave them out. A recording of only CAN traffic, with no ECU online, can be saved too.

Convert Bus Log... in the Logs pane turns a CAN log into one of the other formats: MDF4 (**.mf4**), Vector BLF (**.blf**) or CalTool's text log (**.txt**).

16. Flashing and Diagnostics

16.1. Flashing over XCP

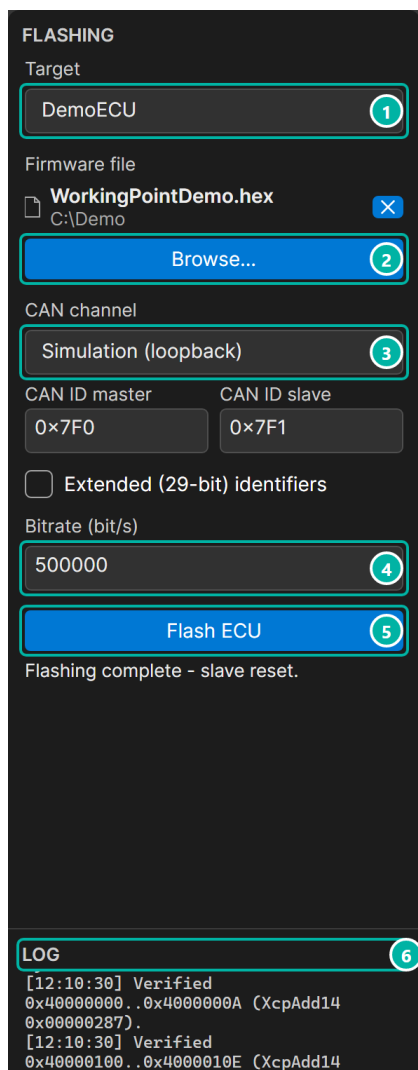


Figure 37. Flash Firmware after a successful flash

1. **Target**—a device from ECU Setup (settings filled in), **Generic XCP**, or **Vector VX flash tool**.
2. **Firmware file**—**.hex**, **.s19** or **.bin** (with base address). A device's memory image is chosen automatically.
3. **CAN channel**—or IP settings for XCP on Ethernet.
4. **Bitrate**—with the CAN identifiers above.
5. **Flash ECU**—**Cancel** stops, but leaves the ECU without working software.
6. **LOG**—every step, including verification.

A device reachable over both CAN and Ethernet is listed once per route. CalTool checks that the firmware matches the device's A2L file, takes the device offline while flashing and online again afterwards.

Vector VX flash tool: select the folder with **VxEcuFlash.exe** and the **.vxp** container, choose the firmware and click **Open VX flash tool**. Start the flash in Vector's window; CalTool shows its progress and result.



Do not disconnect power or the interface while flashing.

16.2. Diagnostics over UDS and KWP2000

The **Diagnostics (UDS)** pane uses the ECU's description file (**.uds**), normally supplied by Altech with

the ECU software. The file says whether the ECU speaks UDS (ISO 14229) or the older KWP2000 (ISO 14230 over CAN, used by most European ECUs from before about 2010); without a file, choose the **Protocol** in the pane.

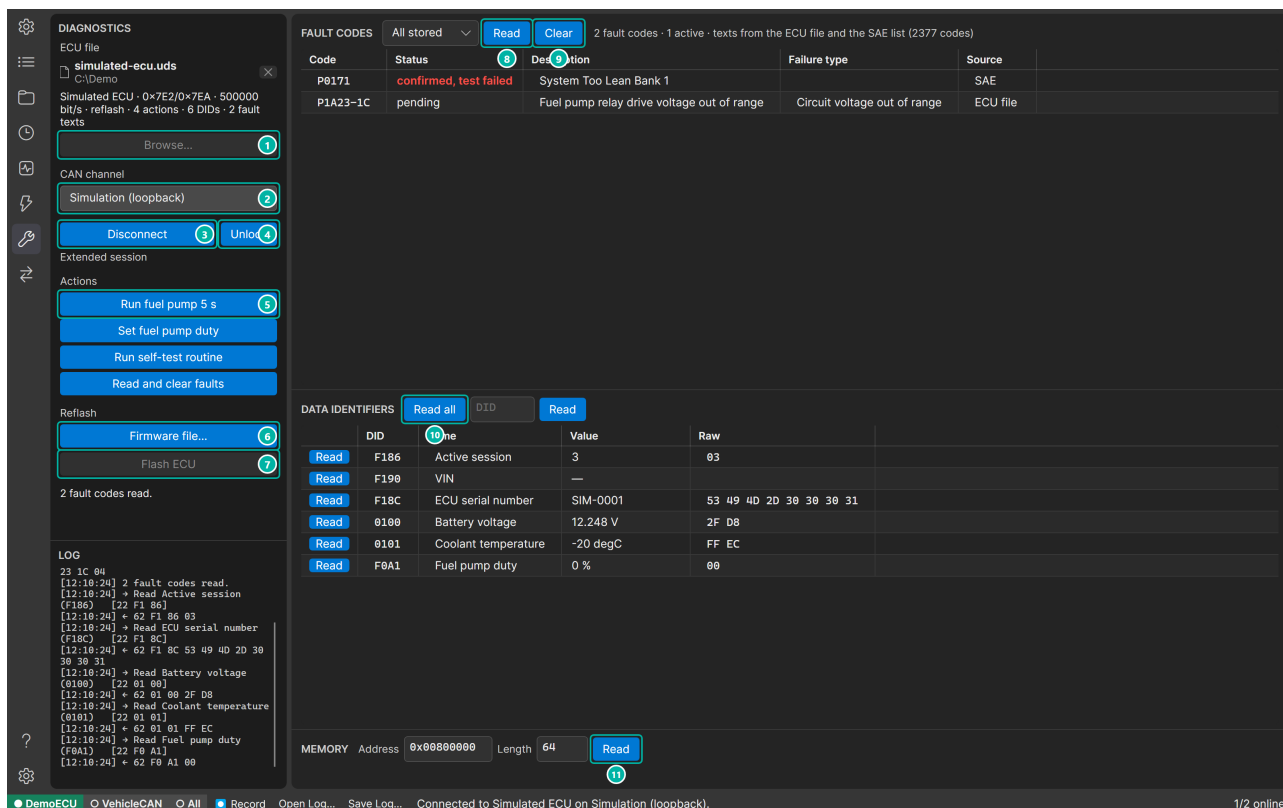


Figure 38. Diagnostics pane connected to the simulated ECU

1. **ECU file** — the .uds file. The line above summarises it.
2. **CAN channel** — identifiers and bitrate come from the file.
3. **Connect / Disconnect** — the diagnostic session.
4. **Unlock** — security access.
5. **Actions** — routines defined in the file, for example running a pump.
6. **Firmware file...** — firmware for the UDS flash sequence.
7. **Flash ECU** — flashes over the bootloader.
8. **Read** — reads fault codes; active ones are red.
9. **Clear** — clears fault codes.
10. **Read all** — reads all data identifiers; type a **DID** to read another.
11. **MEMORY Read** — reads memory at an address.

An action can stop and ask you to do something — open a bleed screw, switch something on — before it goes on: **Continue** or **Stop**. The ECU's session is kept alive while you do.

Fault codes, identifiers and memory can also be read without a file, by entering the CAN identifiers.

Writing an ECU description file is described in [Appendix A](#).

Appendix A: ECU Description Files (.uds)

An ECU description file tells the **Diagnostics (UDS)** pane ([Section 16.2](#)) everything that is specific to one ECU:

- how to reach it: CAN identifiers, bitrate, protocol, session and security access,
- its **actions**: buttons for diagnostic services such as running a pump,
- its **data identifiers** (DIDs), with names, scaling and units,
- the texts for its **manufacturer-specific fault codes**,
- its **flash sequence**.

It is a plain text file, written and edited in any text editor. Keep one file per ECU.

CalTool never assumes a flash sequence: an ECU can only be flashed over UDS with a file that describes how. Reading and clearing fault codes, reading data identifiers and reading memory are standard services and work without a file.

Every common service of the UDS standard (ISO 14229-1) has a named step, so a file reads as what it does: `comm disable-rx-tx normal` rather than the bytes `28 03 01`. [Section A.6](#) shows the steps in use.

A.1. Example

The file below describes CalTool's built-in simulated ECU and uses every part of the format. It can be tried out on the **Simulation** channel without any hardware:

```
title          Simulated ECU

request        0x7E2
response       0x7EA
bitrate        500000

diagnostic-session extended
keepalive      2s
security       01 zeros

# ----- reflash

align         8
fill          0xFF
area          0x00800000 0x01000000
last-segment   0x00800000

flash
  session programming  timeout 10s
```

```

unlock 01 zeros
routine start FF00 44 {start} {size}    timeout 60s
download
routine start  FF01 44 {start} {words} {sum32}    timeout 30s
routine result FF01  ->  {sum32:le}
reset hard
end

# ----- actions
#
# action "<name>" ... end - a button in the pane. The same steps as a reflash, minus
# the image, plus
# {ask:label} for a value typed when the button is pressed.

action "Run fuel pump 5 s"
    say Starting the pump...
    iocontrol F0A1 03 01          # InputOutputControlByIdentifier:
shortTermAdjustment, on
    wait 5s
    iocontrol F0A1 00          # returnControlToECU
    say Pump stopped.
end

action "Set fuel pump duty"
    iocontrol F0A1 03 {ask:duty}    # one byte, asked for when pressed
end

action "Run self-test routine"
    routine start F001          # RoutineControl start, the application's own
routine
    wait 500ms
    routine result F001
end

action "Read and clear faults"
    dtc count                  # how many, before reading them
    dtc read
    dtc clear
    dtc read
end

action "Read identification"
    read F190 F18C              # several identifiers in one request
end

action "Write fuel pump duty"
    write F0A1 {ask:duty}        # WriteDataByIdentifier
    read  F0A1 expect {ask:duty} # read back: must be what was written
end

```

```
action "Silence the bus for 5 s"
  comm disable-rx-tx normal      # application messages off
  dtc off                        # no faults stored meanwhile
  wait 5s
  dtc on
  comm enable-rx-tx normal
end

action "Combine voltage and temperature"
  define-did F300 clear optional  # forget an earlier definition
  define-did F300 by-id 0100 1 2 0101 1 2 # two bytes of each
  read F300
end

action "Write and read back memory"
  write-memory 0x00800100 DE AD BE EF
  read-memory 0x00800100 4 expect DE AD BE EF
end

# ----- data identifiers

did F186 "Active session"      u8
did F190 "VIN"                  ascii
did F18C "ECU serial number"    ascii
did 0100 "Battery voltage"      u16 *0.001 V
did 0101 "Coolant temperature"  s16 degC
did F0A1 "Fuel pump duty"      u8 %

# ----- fault codes

dtc P1A23 "Fuel pump relay stuck"
dtc P1A23-1C "Fuel pump relay drive voltage out of range"
```

A.2. How a file is written

- Each line is a **setting** (key value), a **declaration** (did, dtc), or a **step**.
- Steps are grouped in **blocks**: flash to end, and action "name" to end.
- # starts a comment, except inside quotes.
- Keywords are not case-sensitive.
- **Bytes, identifiers and sub-functions** are written in hexadecimal, in any grouping: 10 02, 1002 and 0x10 0x02 are the same.
- **Addresses, lengths, positions and sizes** are numbers: decimal (16), or hexadecimal with 0x in front (0x00800100).

- **Bitrates** are numbers, and may be written **500k** or **1M**.
- **Durations** are written **3s**, **500ms**, or a plain number of seconds.
- A file does not need every part. A file with only DIDs and fault code texts is fine.

In the tables below, a word in angle brackets, such as **<id>**, stands for a value you write; a part in square brackets, such as **[timeout 5s]**, may be left out. Where a step takes one word out of several, each choice is listed on its own line.

A.3. Connection and session settings

Setting	Meaning	Default
title <name>	The ECU's name, shown in the pane	the file name
protocol uds protocol kwp2000	The ECU's diagnostic protocol (Section A.7)	uds
request <id>	CAN identifier CalTool sends on	entered in the pane
response <id>	CAN identifier the ECU answers on	entered in the pane
extended-ids yes	29-bit CAN identifiers	no
bitrate <bit/s>	CAN bitrate	entered in the pane
can-fd on data-bitrate <bit/s>	CAN FD; data-bitrate also sets the data phase bitrate	entered in the pane
p2 <duration>	How long to wait for an answer	3s
p2* <duration>	How long to wait after the ECU answers "response pending"	10s
retries <n>	How many times to resend after no answer	0
diagnostic-session <session>	The session Connect opens: default , programming , extended , safety , none , or a number	extended (UDS), none (KWP2000)
start-communication no	KWP2000 only: do not send StartCommunication on Connect	yes
keepalive <duration> keepalive off	How often TesterPresent is sent to keep the session open	2s
security <level> <key>	What Unlock does, written like the unlock step (Section A.8). Without this line there is no Unlock button.	none

A setting given in the file replaces the matching field in the pane, so the two cannot disagree.

A.4. Steps

Steps are used inside **flash** and **action** blocks. Each step names a diagnostic service; its parameters are the bytes from the ECU's documentation. The service number is given in brackets.

Session, communication and the link

Step	What it does
session default session programming session extended session safety session <number>	Changes the diagnostic session (0x10). KWP2000 names: standard , programming , development , adjustment .
reset hard reset key-off reset soft reset enable-rapid-power-shutdown reset disable-rapid-power-shutdown	Resets the ECU (0x11)
tester-present tester-present quiet	Sends TesterPresent (0x3E); quiet asks the ECU not to answer
comm <control> <type>	Turns the ECU's own messages on or off (CommunicationControl, 0x28). <control> is one of enable-rx-tx , enable-rx-disable-tx , disable-rx-enable-tx , disable-rx-tx . <type> is normal (application messages), network-management (or nm) or all .
comm enable-rx-disable-tx-enhanced <type> <node> comm enable-rx-tx-enhanced <type> <node>	The same for one node on a sub-bus; <node> is its two-byte identification number
timing read timing read-extended timing default timing set <bytes>	Reads or sets the ECU's timing parameters (AccessTimingParameter, 0x83)
link verify <bitrate> link verify-fixed <id> link verify-specific <bitrate> link transition	Changes the CAN bitrate (LinkControl, 0x87). verify asks whether the ECU can use the bitrate; transition makes the change (see the note below).
start-communication stop-communication	KWP2000 only: opens or closes the conversation (0x81, 0x82)



After **link transition** the ECU talks at the new bitrate, but CalTool's channel stays at the bitrate set in the pane. Use **link transition** only as the last step of a session, or change the pane's bitrate and connect again afterwards.

Security access

Step	What it does
unlock <level> <key>	Requests a seed and sends the key (0x27), see Section A.8
seed <level>	Only requests the seed; the seed is shown in the log. <level> is odd.
key <level> <bytes>	Only sends a key. <level> is even, one above the seed's level.

Step	What it does
authentication <task> [<bytes>]	Certificate-based authentication (0x29). <task> is one of deauthenticate, verify-certificate-unidirectional, verify-certificate-bidirectional, proof-of-ownership, transmit-certificate, request-challenge, verify-proof-unidirectional, verify-proof-bidirectional, configuration. The certificate or proof follows as bytes.

Reading and writing data

Step	What it does
read <id> read <id> <id> ...	Reads one data identifier, or several in one request (0x22)
read-scaling <id>	Reads how the ECU encodes an identifier's value (0x24)
write <id> <bytes>	Writes a data identifier (0x2E)
periodic slow <ids> periodic medium <ids> periodic fast <ids> periodic stop [<ids>]	Asks the ECU to send identifiers repeatedly (0x2A). An identifier is written F201, or as its last byte, 01. The values are shown in the CAN trace.
define-did <id> by-id <source> <position> <size> ...	Defines a new identifier from parts of others (0x2C). Repeat <source> <position> <size> for each part; the first byte of the source is position 1.
define-did <id> by-memory <address> <size> ... [format 14]	Defines a new identifier from memory areas. Repeat <address> <size> for each area.
define-did <id> clear define-did clear	Removes one defined identifier, or all of them
ident <option>	KWP2000 only: reads the ECU identification (0x1A)
read-memory <address> <length> [format 44]	Reads memory (0x23)
write-memory <address> <bytes> [format 44]	Writes the bytes to memory (0x3D). The length is the number of bytes given.
raw <bytes>	Sends exactly these bytes, for a service not listed here

format is the address and length format byte: the right digit is the number of address bytes, the left digit the number of length bytes. 44 is a 4-byte address and a 4-byte length.

Routines and outputs

Step	What it does
routine start <id> [<bytes>] routine stop <id> routine result <id>	Starts or stops a routine, or reads its result (0x31). The bytes after the identifier are the routine's own parameters.

Step	What it does
iocontrol <id> adjust <state> iocontrol <id> return iocontrol <id> freeze iocontrol <id> default	Controls an input or output (0x2F). adjust takes control and sets the state that follows; return gives control back to the ECU; freeze holds the current state; default sets the ECU's default state. The control byte may be written instead of the name (03 for adjust , 00 for return). io may be written instead of icontrol .

Fault codes

Step	What it does
dtc read [<mask>]	Reads stored fault codes (0x19 02)
dtc count [<mask>]	Reads only how many fault codes are stored (0x19 01)
dtc supported	Lists every fault code the ECU can report (0x19 0A)
dtc snapshot <code> [<record>]	Reads the snapshot (freeze frame) stored with a fault code (0x19 04)
dtc snapshot-ids	Lists which fault codes have snapshots (0x19 03)
dtc snapshot-record [<record>]	Reads snapshots by record number (0x19 05)
dtc extended <code> [<record>]	Reads the extended data stored with a fault code, such as occurrence counters (0x19 06)
dtc extended-record <record> dtc extended-supported <record>	Reads an extended data record for every fault code, or lists the fault codes that have it (0x19 16, 0x19 1A)
dtc first-failed dtc first-confirmed dtc recent-failed dtc recent-confirmed	Reads the first or most recent failed or confirmed fault code (0x19 0B to 0E)
dtc permanent	Reads permanent fault codes (0x19 15)
dtc fault-counter	Reads the fault detection counters (0x19 14)
dtc severity [<severity>] [<mask>] dtc severity-count [<severity>] [<mask>] dtc severity-info <code>	Reads fault codes by severity, counts them, or reads one code's severity (0x19 08, 07, 09)
dtc mirror [<mask>] dtc mirror-count [<mask>] dtc mirror-extended <code> [<record>]	The same for the ECU's mirror memory (0x19 0F, 11, 10)
dtc emissions [<mask>] dtc emissions-count [<mask>]	Emissions-related (OBD) fault codes (0x19 13, 12)
dtc user-memory <mask> <memory> dtc user-snapshot <code> <record> <memory> dtc user-extended <code> <record> <memory>	A user-defined fault memory, selected by its number (0x19 17, 18, 19)
dtc wwh-obd <group> [<mask>] [<severity>] dtc wwh-obd-permanent <group> dtc readiness <group> <readiness>	WWH-OBD reports (0x19 42, 55, 56)
dtc report <type> [<bytes>]	Any other fault code report, by its number
dtc clear [<group>] [<memory>]	Clears fault codes (0x14). The group is FFFFFF (all) if left out; <memory> selects a user-defined fault memory.

Step	What it does
<code>dtc on [<bytes>]</code> <code>dtc off [<bytes>]</code>	Turns fault code storing on or off (0x85). Some ECUs want a group after it, for example <code>dtc off FFFFFFFF</code> .

A fault code is written as it is printed, `P0171-1C`, or as six hex digits, `01711C`. A status mask, severity mask or record number left out at the end is `FF`, meaning all. Parameters are written in the order the table shows.

Events and protected data

Step	What it does
<code>response-on-event <event> [store] [<bytes>]</code>	Sets up a message the ECU sends by itself when something happens (ResponseOnEvent, 0x86). <code><event></code> is one of <code>on-dtc-change</code> , <code>on-did-change</code> , <code>on-comparison</code> , <code>report-recent-dtc</code> , <code>report-dtc-records</code> , <code>start</code> , <code>stop</code> , <code>clear</code> , <code>report-events</code> . The bytes are the event's settings from the ECU's documentation; <code>store</code> keeps the event after the ECU is switched off.
<code>secured <bytes></code>	Sends protected data (SecuredDataTransmission, 0x84)

Transfers and files

Step	What it does
<code>download</code>	Flash sequence only: transfers the firmware image to the ECU (0x34, 0x36, 0x37)
<code>request-download <address> <size> [format 44] [data-format 00]</code> <code>request-upload <address> <size> [format 44] [data-format 00]</code>	Starts a transfer to or from the ECU on its own (0x34, 0x35). Use <code>download</code> for a firmware image.
<code>transfer-data <counter> [<bytes>]</code>	Sends one block of a transfer, with its block counter (0x36). For an upload, write no bytes: the ECU's answer is the data.
<code>transfer-exit [<bytes>]</code>	Ends a transfer (0x37)
<code>file add "<path>" <size> [<compressed size>] [data-format 00]</code> <code>file replace "<path>" <size> ...</code> <code>file resume "<path>" <size> ...</code> <code>file read "<path>" [data-format 00]</code> <code>file delete "<path>"</code> <code>file dir "<path>"</code>	Works with a file in the ECU's file system (RequestFileTransfer, 0x38). The data then goes with <code>transfer-data</code> and <code>transfer-exit</code> .

Flow of the action

Step	What it does
<code>wait <duration></code>	Pauses
<code>confirm "<text>"</code>	Shows the text and waits for Continue . Stop cancels the action.
<code>say <text></code>	Writes the text to the log
<code>each region to end</code>	Flash sequence only: repeats the steps inside once for each region of the firmware image

While an action waits in **wait** or **confirm**, the session is kept alive, so a long pause does not end it.

A sub-function or identifier can always be written as a number instead of a name, for example **session 7F** or **comm 03 01**.

Options on a step

These may be added at the end of any step that sends a request:

Option	Meaning
timeout <duration>	Waits this long for the answer instead of p2
expect <bytes>	The answer must contain these bytes, or the step fails. .. stands for any one byte. -> (a minus and a greater-than sign) may be written instead of expect .
no-reply	Does not wait for an answer
quiet	Asks the ECU not to answer if the step works, and does not wait. Only for steps with a sub-function: session , reset , comm , timing , link , authentication , response-on-event , define-did , dtc on and dtc off .
optional	If the step fails, logs it and continues instead of stopping

The bytes after **expect** are compared with the data in the answer, not with the service's echo. For **read F186 expect 03**, the ECU's answer **62 F1 86 03** matches. For **raw** steps the comparison starts at the first byte.

A negative response from the ECU always fails the step (unless it is **optional**). A "response pending" answer is handled automatically, and so is KWP2000's "busy, repeat request".

A.5. Actions

Each **action "<name>"** to **end** block becomes a button in the pane. An action can use every step except **download** and **each region**.

{ask:<label>} in a step asks for a value when the button is clicked, and inserts it as one byte. A width and byte order can be added: **{ask:speed:2}** is two bytes, **{ask:address:4:le}** is four bytes with the least significant byte first. The label is one word, and a label used twice in the same action is asked for once. An identifier can be asked for too, for example **routine start {ask:routine} 01**.

Example: a pump run with a pause for the operator.

```
action "Run fuel pump"
  confirm "Fuel line connected to the test rig? Press Continue to start the
  pump."
  ioccontrol F0A1 adjust 01      # take control: pump on
  wait 30s
  ioccontrol F0A1 return        # give control back
  say Pump stopped.
```

```
end
```

A.6. Examples

The examples below are actions and flash sequences for a typical ECU. The identifiers, routines and addresses are examples: take the real ones from the ECU's diagnostic specification.

Reading identification

Several identifiers are read in one request. **read-scaling** shows how the ECU encodes a value, which helps when writing a **did** line for it.

```
action "Read identification"
  read F190 F18C F191 F195      # VIN, serial number, HW and SW version
end

action "How is the battery voltage encoded?"
  read-scaling 0100
end
```

Writing a value and checking it

write sets a data identifier. Reading it back with **expect** makes the action fail if the ECU did not keep the value.

```
action "Set idle speed target"
  unlock 01 zeros
  write 0120 {ask:rpm:2}
  read 0120 expect {ask:rpm:2}
end
```

Testing an output

Take control of an output, set it, and always give control back. **confirm** gives the operator time to check the result.

```
action "Test cooling fan"
  confirm "Keep hands away from the fan. Press Continue to start it."
  ioccontrol F0B2 adjust 64      # 64 hex = 100 %
  confirm "Is the fan running? Press Continue to stop it."
  ioccontrol F0B2 return
end

action "Hold the fan as it is for 10 s"
```

```
iocontrol F0B2 freeze
wait 10s
iocontrol F0B2 return
end
```

Running a routine

Start the routine, wait for it, then read its result. **expect** after **routine result** checks the routine's own answer: here the first byte must be **00**, meaning passed.

```
action "Injector leak test"
  session extended
  routine start 0203 01          # 01: cylinder 1
  wait 5s
  routine result 0203 expect 00
end

action "Stop injector leak test"
  routine stop 0203
end
```

Working with the fault memory

Count first, then read the codes and the data stored with one of them. A snapshot is the freeze frame recorded when the fault occurred; extended data holds counters such as how often it occurred.

```
action "Fault overview"
  dtc count                # how many are stored
  dtc read 08              # only confirmed faults (status bit 3)
  dtc permanent
  dtc first-confirmed
  dtc recent-confirmed
end

action "Details of a fault"
  dtc snapshot P0171-1C    # all snapshots of this fault
  dtc extended P0171-1C 01 # extended data record 01
end

action "What can this ECU report?"
  dtc supported
  dtc snapshot-ids
  dtc fault-counter
end

action "Clear emissions-related faults"
```

```
dtc clear FFFF33
end
```

Keeping the bus quiet during a test

Before a test that provokes faults, stop fault code storing and silence the ECU's own messages, so other ECUs do not react. Always turn both back on at the end.

```
action "Sensor plausibility test"
  dtc off
  comm disable-rx-tx normal
  confirm "Disconnect the coolant temperature sensor, then press Continue."
  read 0101                      # the substitute value the ECU uses now
  confirm "Reconnect the sensor, then press Continue."
  comm enable-rx-tx normal
  dtc on
  dtc read                      # nothing should have been stored
end
```

Combining values into one identifier

define-did builds a new identifier from parts of others, so several values come back in one answer. Here identifier F201 gets the two bytes of the battery voltage and the two bytes of the coolant temperature. **periodic** then asks the ECU to send it repeatedly; the values appear in the CAN trace ([Chapter 15](#)). Only identifiers F200 to F2FF can be sent periodically.

```
action "Start fast monitoring"
  define-did F201 clear optional      # remove an earlier definition
  define-did F201 by-id 0100 1 2  0101 1 2
  read F201
  periodic fast F201
end

action "Stop fast monitoring"
  periodic stop
  define-did F201 clear
end

action "Monitor a memory variable"
  define-did F202 by-memory 0x40001A00 4
  periodic medium F202
end
```

Reading and writing memory

write-memory takes the bytes to write; the length is counted for you. Reading the memory back with **expect** checks the write.

```
action "Patch test value"
  unlock 01 zeros
  write-memory 0x40001A00 00 00 01 F4
  read-memory 0x40001A00 4 expect 00 00 01 F4
end

action "Read 16 bytes at an address"
  read-memory {ask:address:4} 16
end
```

Security access by hand

seed and **key** do one half of security access each. With **optional**, a level the ECU does not have is logged and the action goes on, which shows which levels exist. A key calculated outside CalTool can be typed in when the button is clicked.

```
action "Which security levels exist?"
  seed 01 optional
  seed 03 optional
  seed 05 optional
  seed 11 optional
end

action "Unlock with a typed key"
  seed 03 # the seed is shown in the log
  key 04 {ask:key:4}
end
```

Messages from the ECU on an event

response-on-event makes the ECU send a message by itself, for example each time a fault code changes status. The bytes after the event type come from the ECU's documentation: here the event window (02), the status mask (09) and the request to answer with (19 02 09).

```
action "Report fault changes"
  response-on-event on-dtc-change 02 09 19 02 09
  response-on-event start
end

action "Stop fault change reports"
```

```
response-on-event stop
response-on-event clear
end
```

Reading a file from the ECU

An ECU with a file system sends a file after **file read**, one block per **transfer-data**. The counter starts at **01**. The data is shown in the log.

```
action "Read crash log"
  session extended
  unlock 01 zeros
  file read "/log/crash.bin"
  transfer-data 01
  transfer-data 02
  transfer-exit
end

action "Delete crash log"
  file delete "/log/crash.bin"
end
```

Reading a memory area as a transfer

Some ECUs give larger memory areas only as an upload. The answer to **request-upload** says how large each block is.

```
action "Upload calibration area"
  session programming
  unlock 01 zeros
  request-upload 0x00A00000 0x100
  transfer-data 01
  transfer-exit
  reset hard
end
```

Checking the timing and the bitrate

```
action "Read timing parameters"
  timing read
end

action "Can the ECU run at 1 Mbit/s?"
  link verify 1M
```

end

A complete flash sequence

A flash sequence usually prepares the ECU first: fault code storing is stopped and its messages are silenced, so the flash does not leave faults behind in the vehicle. **quiet** is used because no answer is needed. After the reset, everything is turned back on.

```
request    0x7E0
response   0x7E8
bitrate    500000

flash
  # Prepare
  session extended
  dtc off                                quiet
  comm disable-rx-tx normal             quiet

  # Program
  session programming timeout 10s
  unlock 01 dll SeedKey.dll
  each region
    routine start FF00 44 {start} {size} timeout 60s    # erase
    download
    routine start 0202 {crc32} expect 00                # check this region
  end
  routine start FF01                                expect 00    # check dependencies
  reset hard

  # Restore
  session extended timeout 10s
  comm enable-rx-tx normal
  dtc on
  dtc clear
end
```

Finding out what an unknown ECU supports

With **optional** on each step, a refusal is logged and the action carries on. The log then shows which services and identifiers the ECU answers.

```
action "Probe services"
  session extended          optional
  read F186                 optional
  read F18C                 optional
```

```

dtc count          optional
dtc supported      optional
timing read        optional
read-memory 0x00000000 4 optional
end

```

A.7. KWP2000 ECUs

With **protocol kwp2000** the steps are written exactly as above, but CalTool sends the KWP2000 (ISO 14230-3) form of each service. The main difference is how an ECU's objects are addressed: KWP2000 uses a **one-byte local identifier** where UDS uses a two-byte identifier.

Step	UDS sends	KWP2000 sends
read 80	22 00 80	21 80
read F190	22 F1 90	22 F1 90
write 80 01	2E 00 80 01	3B 80 01
iocontrol 80 adjust 01	2F 00 80 03 01	30 80 07 01
routine start F3 01	not available	31 F3 01
routine stop F3	not available	32 F3
routine result F3	not available	33 F3
routine start FF00 01	31 01 FF 00 01	31 01 FF 00 01
dtc read	19 02 FF	17 FF 00
dtc clear	14 FF FF FF	14 FF 00
read-memory 0x123456 16	32-bit address and length	23 12 34 56 10 (3-byte address, at most 255 bytes)
write-memory 0x123456 01 02	32-bit address and length	3D 12 34 56 02 01 02
seed 01	27 01	27 01
tester-present	3E 00	3E
session standard	not available	10 81
session programming	10 02	10 85
session C0	10 C0	10 C0

In short: write a one-byte identifier for a local identifier and a two-byte identifier for a common identifier. **did 80** "..." lines and the pane's identifier box take two hex digits in the same way.

Connect sends StartCommunication first, and then a session request only if **diagnostic-session** names one (for example **diagnostic-session C0** for a PSA ECU). **Disconnect** sends StopCommunication. Fault codes come back as two-byte codes with the ECU's own status byte, which is shown as a number.

For KWP2000 I/O control the names follow KWP2000's numbering: **adjust** is **07**, **return** is **00**, **report** is **01** (report the current state), **default** is **04**, **freeze** is **05** and **long-term** is **08**.

The steps that exist only in UDS, or that KWP2000 lays out differently, are refused in a KWP2000 file: `comm`, `timing`, `link`, `authentication`, `read` with several identifiers, `read-scaling`, `periodic`, `define-did`, `response-on-event`, `request-download`, `request-upload`, `transfer-data` and `file`. Write such a request with `raw`.

A.8. Security access

`unlock <level> <key>` (and the `security` setting) requests a seed with sub-function `<level>` and sends the key with the next sub-function (`<level> + 1`). The key is one of:

Key	Meaning
<code>zeros</code>	A key of zeros, as long as the seed
<code>seed-only</code>	Only requests the seed
<code>constant <bytes></code>	Always sends these bytes
<code>xor <bytes></code>	The seed XOR-ed with these bytes
<code>dll <file></code>	Calculates the key with the ECU manufacturer's seed & key DLL

The DLL path is relative to the `.uds` file. Both 32-bit and 64-bit DLLs are supported. If the DLL needs other files, keep them in the same folder.

Two kinds of DLL are supported:

- The common interface `int UDSComputeKeyFromSeed(accessMode, seedLength, seed, keylength, key)`. If the DLL's function has another name, add `func <name>`; add `cdecl` if it uses that calling convention.
- The Vector interface used by CANape and CANoe: write `func GenerateKeyEx`.

Example: `security 03 dll SeedKey.dll func GenerateKeyEx`

If the ECU answers with a seed of only zeros, it is already unlocked and no key is sent.

A.9. Data identifiers

```
did <id> "<name>" [<type>] [le] [*<factor>] [+<offset>] [<unit>]
```

Each line adds a row to the pane's data identifier table.

Part	Meaning
<code><id></code>	The identifier, four hex digits (two for a KWP2000 local identifier)
<code><type></code>	<code>u8</code> , <code>u16</code> , <code>u32</code> (unsigned), <code>s8</code> , <code>s16</code> , <code>s32</code> (signed), <code>f32</code> (floating point), <code>ascii</code> (text) or <code>hex</code> (the raw bytes). Default <code>hex</code> .
<code>le</code>	Least significant byte first. Default is most significant byte first.

Part	Meaning
*<factor>, +<offset>	Scaling: value = raw × factor + offset
<unit>	The unit shown after the value; may be several words

Example: did 0100 "Battery voltage" u16 *0.001 V shows the raw value 2F D8 as 12.248 V.

A.10. Fault code texts

```
dtc <code> "<text>"
```

The text is shown when the ECU reports the code. **P1A23** gives a text for every failure type of that code; **P1A23-1C** gives a text for one failure type only. The six-digit hexadecimal form (**1A231C**) may also be used.

Standard codes (P0xxx, P2xxx, P34xx, B0xxx, C0xxx, U0xxx, U3xxx) are already known to CalTool and need no line, but a **dtc** line replaces the built-in text. Manufacturer codes (for example P1xxx) need a line to get a text.

A.11. The flash sequence

The **flash to end** block runs when **Flash ECU** is clicked, using the chosen firmware file. Steps outside any block are also treated as part of the flash sequence.

A typical sequence opens the programming session, unlocks, erases, downloads, checks and resets:

```
flash
  session programming   timeout 10s
  unlock 01 zeros
  routine start FF00 44 {start} {size}   timeout 60s   # erase
  download
  routine start FF01 44 {start} {size} {crc32}          # check
  reset hard
end
```

To erase, download and check each region of the image separately, put those steps in an **each region** block:

```
flash
  session programming
  unlock 01 zeros
  each region
    routine start FF00 44 {start} {size}   timeout 60s
    download
```

```

    routine start FF01 44 {start} {size} {crc16}
  end
  reset hard
end

```

A longer example, which also prepares and restores the ECU around the flash, is in [Section A.6](#).

Placeholders

A `{name}` in a step is replaced with a value calculated from the firmware image. Inside an **each region** block it describes the current region; outside it describes the whole image, with the gaps between regions filled with the **fill** byte.

Placeholder	Value
<code>{start}</code>	First address
<code>{end}</code>	Address after the last byte
<code>{size}</code>	Number of bytes
<code>{words}</code>	Number of 32-bit words ($\text{size} \div 4$)
<code>{sum32}</code>	Sum of the 32-bit words (most significant byte first)
<code>{sum16}</code>	Sum of the 16-bit words (most significant byte first)
<code>{sumbytes}</code>	Sum of the bytes
<code>{crc16}</code>	CRC-16/CCITT-FALSE
<code>{crc16-xmodem}</code> , <code>{crc16-arc}</code> , <code>{crc16-modbus}</code> , <code>{crc16-kermit}</code> , <code>{crc16-mcrf4xx}</code>	Other common CRC-16 variants
<code>{crc32}</code>	CRC-32 (as used by zip and Ethernet)
<code>{crc32-bzip2}</code> , <code>{crc32-mpeg2}</code> , <code>{crc32-posix}</code> , <code>{crc32-jamcrc}</code> , <code>{crc32c}</code> , <code>{crc32-autosar}</code>	Other common CRC-32 variants
<code>{index}</code>	Number of the current region
<code>{count}</code>	Number of regions

Values are four bytes, most significant byte first, except the 16-bit checksums, which are two bytes. Add a size or **le** to change this: `{size:2}`, `{sum32:le}`, `{crc16:le}`.

download always transfers each region separately, never the gaps between them.

Image settings

Setting	Meaning	Default
<code>align <bytes></code>	Pads each region to a multiple of this many bytes	8
<code>fill <byte></code>	The byte used for padding and gaps (the value of erased flash)	0xFF
<code>area <start> <end></code>	Refuses a firmware image with data outside this address range	not checked
<code>sector <start> <size></code>	A flash sector. Repeat the line for each sector; the image is split into one region per sector, in the order listed.	the image's own regions
<code>last-segment <address></code>	Flashes the region at this address last (for example a boot header)	none
<code>address-format <byte></code>	The address and length format byte for RequestDownload; 0x00 leaves it out	0x44
<code>data-format <byte></code>	The data format byte for RequestDownload	0x00
<code>max-block <bytes></code>	Largest number of data bytes per TransferData	set by the ECU

A.12. Writing a file for a new ECU

1. Start from the example above or another ECU's file.
2. Set `title`, `protocol`, `request`, `response` and `bitrate` from the ECU's diagnostic specification.
3. Add `did` lines for the identifiers you want to read, and `dtc` lines for the manufacturer's fault codes.
4. Add `action` blocks for the services a bench needs. Put a `confirm` before any step that moves something, give back every output an action takes control of, and turn on again what it turns off.
5. Write the `flash` block from the bootloader's specification, and set the image settings.
6. Test the file with **Connect**, **Read** and **Read all** first. Try the flash sequence on a bench ECU before using it in a vehicle.

The pane reports a problem in the file with its line number when the file is loaded.

Appendix B: Computed Channels

A **computed channel** is a signal calculated by a short script from other signals — for example a filter, a power from speed and torque, or a counter. It is plotted, read out and saved like any other signal, under the device name **Calc** (for example **Calc/CoolantTemp_Filt**). It is calculated over an open log, and live during a recording as long as its inputs are logged.

B.1. Creating a computed channel

In the **Files & Calibration Data** pane, click **New...** under **COMPUTED CHANNELS**.

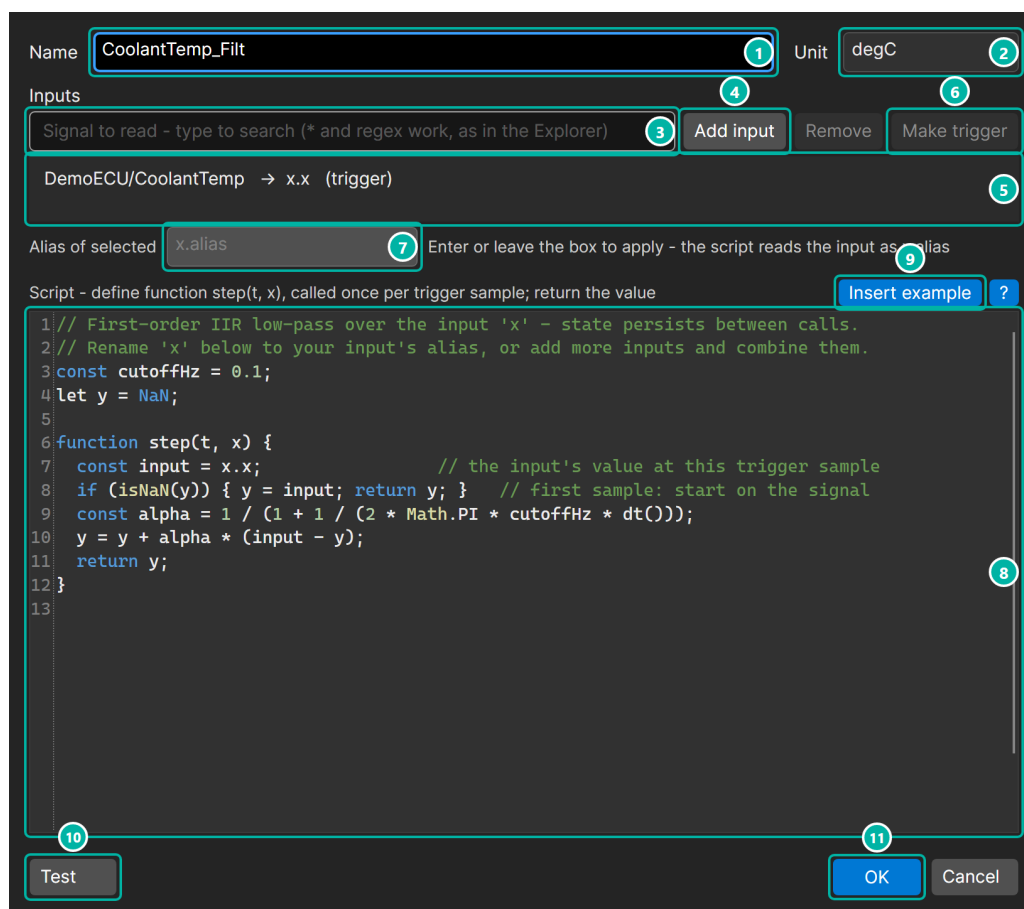


Figure 39. The computed channel editor

1. **Name**
2. **Unit**
3. **Signal** — search for an input: a measurement, log signal or computed channel.
4. **Add input / Remove**
5. **Inputs** — each input and the name the script uses for it (after the arrow).
6. **Make trigger** — the script runs once per sample of the trigger input.
7. **Alias of selected** — renames an input for the script.
8. **Script** — JavaScript (see below).

9. **Insert example** — a low-pass filter. ? summarises what a script can use.
10. **Test** — runs the script over the log or recording and reports the result.
11. **OK** — saves the channel.

To use the channel, drag **Calc/<name>** from the Explorer's **LOG SIGNALS** section onto a plot, or from the COMPUTED CHANNELS list onto a page or plot.

B.2. How the script works

A script defines one function, **step**:

```
function step(t, x) {
  return x.speed * 3.6;    // m/s to km/h
}
```

- **step** is called **once for every sample of the trigger input** (the first input, unless another is made the trigger). The value it returns is the channel's value at that time.
- **t** is the time in seconds.
- **x.<alias>** is an input's value at that time. For inputs other than the trigger, it is their most recent value.
- Variables declared **outside step** keep their value between calls. This is what makes filters, integrators and counters possible.

A script also has these helpers:

Helper	Returns
dt()	Seconds since the previous trigger sample (NaN for the first sample)
first()	true for the first sample
prev('alias', n)	The input's value n samples ago
prevOut(n)	This channel's own value n samples ago
changed('alias')	true if the input changed since the previous sample
posEdge('alias', level)	true when the input rises above level
negEdge('alias', level)	true when the input falls below level
Math.*	JavaScript's maths functions, such as Math.abs and Math.sqrt

If the script fails for a sample, that sample has no value and the first error is reported.

B.3. Examples

Low-pass filter

The example inserted by **Insert example**. It smooths the input **x** with a cutoff frequency set in hertz:

```
const cutoffHz = 2.0;
let y = NaN;

function step(t, x) {
  const input = x.x;
  if (isNaN(y)) { y = input; return y; } // start on the first value
  const alpha = 1 / (1 + 1 / (2 * Math.PI * cutoffHz * dt()));
  y = y + alpha * (input - y);
  return y;
}
```

A lower **cutoffHz** gives a smoother, slower signal. The red line in [Chapter 11](#) is this script with a cutoff of 0.1 Hz on **CoolantTemp**.

Moving average

The average of the last 21 samples:

```
function step(t, x) {
  let sum = 0;
  for (let n = 0; n <= 20; n++) sum += prev('x', n);
  return sum / 21;
}
```

The first 20 values are empty, because there are not yet enough samples.

Power from speed and torque

Two inputs, with aliases **n** (engine speed in rpm) and **M** (torque in Nm):

```
function step(t, x) {
  return x.n * x.M / 9549; // kW
}
```

Make the input with the faster raster the trigger, so the channel has as many values as possible.

Fuel used

Integrates a fuel flow in litres per hour into litres:

```
let fuel = 0;

function step(t, x) {
  if (!first()) fuel += x.flow * dt() / 3600;
  return fuel;
}
```

Counting events

Counts how many times the input **ignition** switches on:

```
let count = 0;

function step(t, x) {
  if (posEdge('ignition', 0.5)) count++;
  return count;
}
```

Appendix C: CAN Stimulation: Counters, Checksums and Scripts

Many ECUs only accept a CAN message when it carries a rolling counter and a valid checksum, and a test often needs more than a fixed frame: a value that follows another message, a reply to a request, a substitute when a sender goes quiet. The **CAN Trace** pane offers two ways to do this:

- **Counters and checksums in the transmit panel** — no code. A signal of a DBC message is filled in automatically on every send ([Section C.2](#)).
- **Scripts** — short JavaScript programs that send, receive and react to frames on one or more interfaces ([Section C.3](#)).



Counters, checksums and scripts put real frames on the bus, and valid ones are exactly what an ECU acts on. Only run them on a bench or vehicle that is safe to control.

C.1. The example messages

The examples in this appendix use a small CAN database with four messages. Any database with a counter and a checksum signal works the same way; the names below are only what the examples refer to.

Message	ID	Contents
EngineStatus	0x110	EngSpeed, CoolantTemp, ThrottlePos, Gear — received from the bus
BrakeStatus	0x120	BrakePressed, BrakePressure — received from the bus
VCU_TorqueReq	0x200	Crc (byte 0, E2E Profile 1), AliveCounter (0-14), TorqueEnable, TorqueReq — sent by you
GW_EngineInfo	0x300	Checksum (byte 0, CRC-8), Counter (0-15), EngSpeed_GW, CoolantTemp_GW, BrakePressed_GW — sent by you

Load the database on the interface with **Browse...** before connecting. Some examples need a second interface, for example to send a request that a script answers: click + **Add interface** and connect it to the same bus. On the **Simulation** channel every interface is on one simulated bus, so this can be tried without hardware.

C.2. Counters and checksums in the transmit panel

When a message is chosen from the DBC file in the transmit panel ([Section 15.2](#)), every signal has a **Fill** setting.

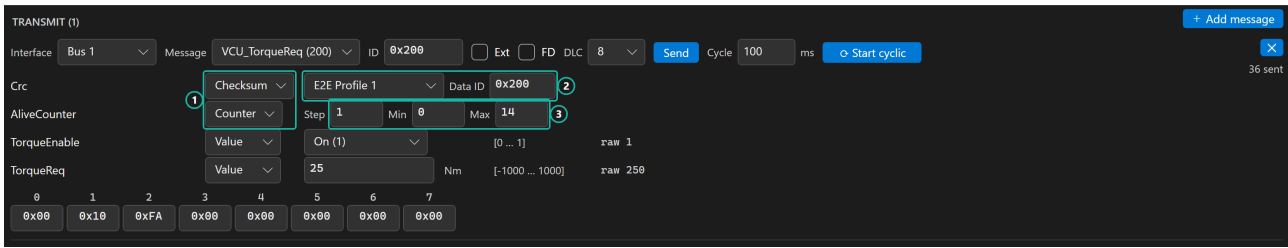


Figure 40. VCU_TorqueReq with a counter and an E2E checksum

1. **Fill — Value** (what you type), **Counter** or **Checksum**.
2. **Checksum** — the algorithm and, for the E2E profiles, the **Data ID**.
3. **Counter** — **Step**, **Min** and **Max**.

To send **VCU_TorqueReq** the way an ECU expects it:

1. Choose **VCU_TorqueReq** in **Message**.
2. For **AliveCounter**, choose **Counter** and set **Max** to **14**.
3. For **Crc**, choose **Checksum**, then **E2E Profile 1**, and enter **0x200** as the **Data ID**.
4. Set **TorqueEnable** and **TorqueReq**, and click **Start cyclic**.

On every send the counter's value is written first, then the checksum is calculated over the finished frame. The counter starts at **Min**, goes up by **Step** and returns to **Min** after **Max**.

The data bytes in the panel keep showing the frame **before** the counter and checksum are filled in. The trace shows the frames that were actually sent.

Counters and checksums need a message from the DBC file. For a raw frame, use a script ([Section C.3.3](#), "Checksum by hand").

Checksum algorithms

Algorithm	Bits	What it covers
CRC-8 SAE J1850	8	All other bytes of the frame
CRC-8H2F	8	All other bytes of the frame
CRC-16 CCITT-FALSE	16	All other bytes of the frame
CRC-32	32	All other bytes of the frame
CRC-32P4	32	All other bytes of the frame
E2E Profile 1	8	The Data ID (low byte, then high byte) and all other bytes. Counter 0-14.
E2E Profile 2	8	All other bytes, then the Data ID for the current counter value. Counter 0-15.
E2E Profile 5	16	All other bytes, then the Data ID (low byte, then high byte). CRC stored low byte first. Counter 0-255.
E2E Profile 11	8	As Profile 1. Counter 0-14.

"All other bytes" means every byte of the frame except the checksum's own, in order. The E2E profiles follow the AUTOSAR E2E library for their default layout: the checksum at the start of the frame and the counter right after it.

For **E2E Profile 2**, the **Data ID** is either one value or a list of 16 values separated by commas, one for each counter value.

C.3. Scripts

Scripts are listed in the **SCRIPTS** tab of the panel below the trace.

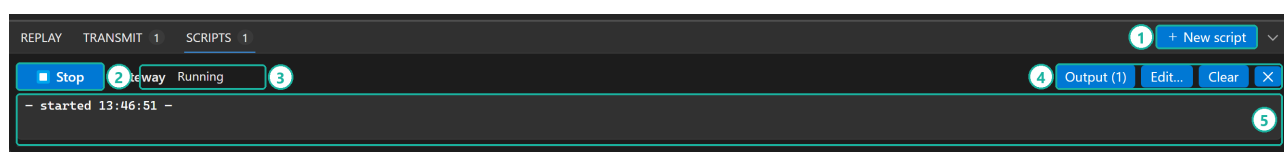


Figure 41. The **SCRIPTS** panel with one script running

1. **+ New script** — opens the script editor.
2. **Run / Stop**
3. **Status** — **Running**, **Stopped**, or the reason the script stopped.
4. **Output** shows what the script printed. **Edit...** opens the editor. **Clear** empties the output. **×** removes the script.
5. **Output** — the newest 500 lines.

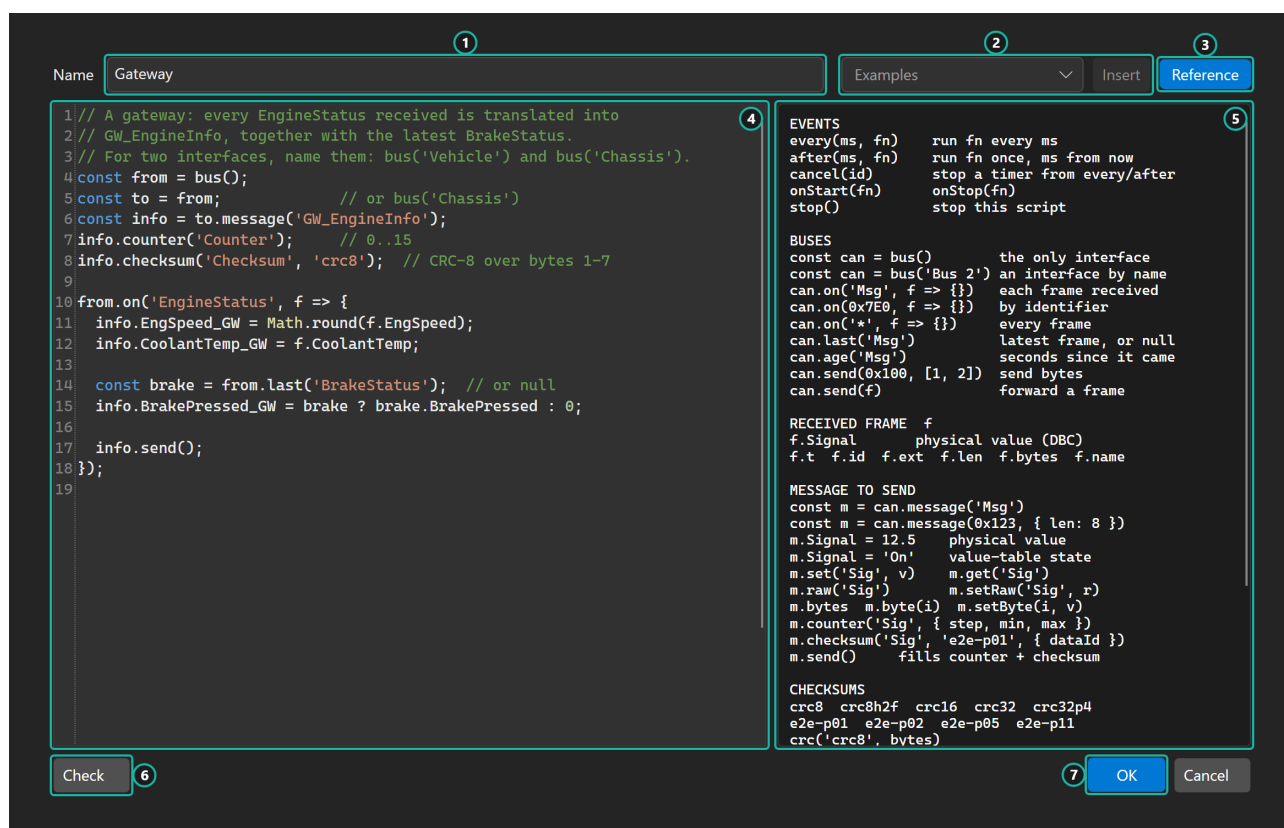


Figure 42. The script editor

1. **Name**
2. **Examples** — choose one and click **Insert** to replace the script with it.
3. **Reference** — shows or hides the reference column.
4. **Script**
5. **Reference column** — everything a script can use.
6. **Check** — finds syntax errors without running the script.
7. **OK** — saves the script. Changes take effect the next time it runs.

A script runs only when you click **Run**. The interfaces it uses must be connected. It stops when you click **Stop**, when one of those interfaces disconnects, or when it fails. On a failure, the status and the output show the reason and the line number.

Scripts are saved with the workspace. Opening a workspace never starts a script.

How a script works

A script is not a loop. When you click **Run**, the script is read from top to bottom once. That first pass sets things up: it picks the interfaces, prepares the messages and says what should happen when. After that, the script's functions are called as things happen:

- `every(10, fn)` calls `fn` every 10 ms, `after(500, fn)` once after 500 ms.
- `can.on('EngineStatus', fn)` calls `fn` for every `EngineStatus` received.
- `onStart(fn)` and `onStop(fn)` run when the script starts and stops.

Variables declared at the top keep their values between calls. Only one function runs at a time, so a script never has to wait for or lock anything.

```
const can = bus();           // the interface
const req = can.message('VCU_TorqueReq'); // a message to send
req.TorqueEnable = 'On';     // set a signal by its state name

let n = 0;                   // keeps its value between calls
every(100, () => {           // every 100 ms
  req.TorqueReq = n++ % 50;  // physical value, in Nm
  req.send();
});
```

Numbers can be written in decimal or, with `0x`, in hexadecimal. Signal values are physical values (with the DBC's factor and offset). A value that does not fit a signal's bits stops the script instead of being cut off.

Reference

Events	Does
<code>every(ms, fn)</code>	Calls <code>fn</code> every <code>ms</code> milliseconds. Returns an id.
<code>after(ms, fn)</code>	Calls <code>fn</code> once, <code>ms</code> milliseconds from now. Returns an id.
<code>cancel(id)</code>	Stops a timer.
<code>onStart(fn) / onStop(fn)</code>	Calls <code>fn</code> when the script starts or stops.
<code>stop()</code>	Stops the script (its <code>onStop</code> functions still run).

Interfaces	Does
<code>bus()</code>	The interface, when there is only one.
<code>bus('Bus 2')</code>	The interface with that name (as on its card).
<code>can.on('EngineStatus', f => {...})</code>	Calls the function for every <code>EngineStatus</code> received.
<code>can.on(0x7E0, f => {...})</code>	The same, by identifier. { <code>id: 0x10</code> , <code>ext: true</code> } for a 29-bit identifier below 0x800.
<code>can.on('*', f => {...})</code>	For every frame received.
<code>can.last('EngineStatus')</code>	The latest <code>EngineStatus</code> received, or <code>null</code> .
<code>can.age('EngineStatus')</code>	Seconds since the latest <code>EngineStatus</code> arrived (<code>Infinity</code> if never).
<code>can.send(0x123, [1, 2, 3])</code>	Sends bytes. Add { <code>ext: true</code> } for a 29-bit identifier.
<code>can.send(f)</code>	Forwards a received frame unchanged.
<code>can.message('VCU_TorqueReq')</code>	A message to send, from the interface's DBC file.
<code>can.message(0x123, { len: 8 })</code>	A message without a DBC description.

A received frame <code>f</code>	Is
<code>f.EngSpeed</code>	A signal's physical value (with a DBC file). Also in <code>f.signals</code> .
<code>f.bytes</code>	The data bytes, as an array.
<code>f.id, f.ext, f.len</code>	Identifier, 29-bit or not, number of bytes.
<code>f.name</code>	The message name, or <code>null</code> .
<code>f.t</code>	Receive time in seconds, as in the trace's Time column.
<code>f.bus</code>	The interface's name.

A message to send <code>m</code>	Does
<code>m.TorqueReq = 12.5</code>	Sets a signal's physical value.
<code>m.TorqueEnable = 'On'</code>	Sets a signal by its state name from the DBC value table.
<code>m.TorqueReq</code>	Reads the value the signal holds now.
<code>m.set('Name', v) / m.get('Name')</code>	The same, for a signal whose name clashes with one of these functions.
<code>m.raw('Name') / m.setRaw('Name', r)</code>	A signal's raw value.

A message to send <code>m</code>	Does
<code>m.bytes, m.byte(i), m.setByte(i, v)</code>	The data bytes.
<code>m.counter('AliveCounter', { step, min, max })</code>	Fills in a counter on every send. Defaults: step 1, min 0, max the largest value that fits.
<code>m.checksum('Crc', 'e2e-p01', { dataId: 0x200 })</code>	Fills in a checksum on every send, after the counters.
<code>m.reset()</code>	Starts the counters from their minimum again.
<code>m.send()</code>	Sends the message. Returns <code>false</code> if it could not be sent.

Without a DBC description, name a counter or checksum by its position: `{ byte: 1, bit: 0, bits: 4 }` is the low four bits of byte 1. Add `bigEndian: true` for a value stored high byte first.

Checksums and ECU values	Does
<code>'crc8', 'crc8h2f', 'crc16', 'crc32', 'crc32p4'</code>	Checksum names (Section C.2.1).
<code>'e2e-p01', 'e2e-p02', 'e2e-p05', 'e2e-p11'</code>	E2E profile names.
<code>crc('crc8', bytes)</code>	The checksum of an array of bytes. A custom CRC: <code>crc({ width: 16, poly: 0x1021, init: 0xFFFF, refIn: false, refOut: false, xorOut: 0 }, bytes)</code> .
<code>e2eCrc('p01', bytes, { dataId: 0x123 })</code>	The E2E checksum of a frame whose checksum is in byte 0. Add <code>crcByte</code> if it is elsewhere.
<code>measurement('DemoECU/EngineSpeed')</code>	A live ECU measurement, or <code>null</code> when it is not being received. <code>measurement('EngineSpeed')</code> works when only one device has that name.
<code>calibration('DemoECU/WarmupEnrichment')</code>	A calibration value: a number, or an array for a curve or map.

Other	Does
<code>now()</code>	Seconds since the script started.
<code>print(...)</code>	Writes to the script's output.
<code>hex(n, digits)</code>	<code>hex(26, 2)</code> is <code>'0x1A'</code> .
<code>Math.*</code>	JavaScript's maths functions, such as <code>Math.round</code> and <code>Math.sin</code> .

Measurements and calibration values can only be read. A script cannot change a calibration.

Examples

All of these are in the editor's **Examples** list and use the messages in Section C.1.

Counter and E2E checksum

Sends `VCU_TorqueReq` every 10 ms, with the counter and checksum filled in as in Section C.2:

```
const can = bus();
```

```
const req = can.message('VCU_TorqueReq');

req.counter('AliveCounter', { max: 14 });           // 0, 1, ... 14, 0, ...
req.checksum('Crc', 'e2e-p01', { dataId: 0x200 }); // computed on every send

req.TorqueEnable = 'On';

every(10, () => {
  req.TorqueReq = 50 * Math.sin(now() * 0.5); // a ramp, -50..50 Nm
  req.send();
});
```

Gateway

Every **EngineStatus** received is translated into **GW_EngineInfo**, together with the latest **BrakeStatus**. On a real bench, receive on one interface and send on another: **const from = bus('Vehicle'), to = bus('Chassis');**.

```
const from = bus();
const to = from;           // or bus('Chassis')
const info = to.message('GW_EngineInfo');
info.counter('Counter');   // 0..15
info.checksum('Checksum', 'crc8'); // CRC-8 over bytes 1-7

from.on('EngineStatus', f => {
  info.EngSpeed_GW = Math.round(f.EngSpeed);
  info.CoolantTemp_GW = f.CoolantTemp;

  const brake = from.last('BrakeStatus'); // or null
  info.BrakePressed_GW = brake ? brake.BrakePressed : 0;

  info.send();
});
```

GW_EngineInfo goes out as soon as each **EngineStatus** has arrived. The trace's Time column shows how long after.

To forward a message unchanged, use **to.send(f)**. To send only when a value changes, keep the previous value in a variable and compare.

Respond to a request

Answers **02 10 xx** on 0x7E0 with a reply on 0x7E8. Raw identifiers and bytes, no DBC needed:

```
const can = bus();
```

```
can.on(0x7E0, f => {  
  if (f.bytes[0] === 0x02 && f.bytes[1] === 0x10) {  
    can.send(0x7E8, [0x06, 0x50, f.bytes[2], 0x00, 0x32, 0x01, 0xF4, 0xAA]);  
    print('answered session request', hex(f.bytes[2], 2));  
  }  
});
```

To try it, add a second interface (**Bus 2**) on the same bus, change `bus()` to `bus('Bus 1')`, run the script, and send `0x02 0x10 0x03` on 0x7E0 from the transmit panel on **Bus 2**.

Timeout fallback

Watches `VCU_TorqueReq` and, while it is missing, sends a safe replacement itself:

```
const can = bus('Bus 1');  
const safe = can.message('VCU_TorqueReq');  
safe.counter('AliveCounter', { max: 14 });  
safe.checksum('Crc', 'e2e-p01', { dataId: 0x200 });  
safe.TorqueEnable = 'Off';  
safe.TorqueReq = 0;  
let missing = false;  
  
every(10, () => {  
  const lost = can.age('VCU_TorqueReq') > 0.1; // in seconds  
  if (lost !== missing) {  
    print(lost ? 'VCU_TorqueReq lost' : 'VCU_TorqueReq back');  
    missing = lost;  
  }  
  if (lost) safe.send();  
});
```

To try it, send `VCU_TorqueReq` cyclically from the transmit panel on a second interface on the same bus, then click **Stop** there: the script takes over.

Send an ECU measurement

Puts a value the ECU measures onto the bus. With the demo ECU of this guide online and `EngineLoad` logged ([Section 9.1](#)), its engine load becomes the torque request:

```
const can = bus();  
const req = can.message('VCU_TorqueReq');  
req.counter('AliveCounter', { max: 14 });  
req.checksum('Crc', 'e2e-p01', { dataId: 0x200 });  
req.TorqueEnable = 'On';
```

```
every(20, () => {  
  const load = measurement('EngineLoad');    // null while not received  
  if (load === null) return;  
  req.TorqueReq = load * 2;                    // 0-100 % as 0-200 Nm  
  req.send();  
});
```

With several ECUs, name the device too: `measurement('DemoECU/EngineLoad')`.

Checksum by hand

A frame without a DBC description. The counter and the E2E checksum are calculated into a plain array of bytes:

```
const can = bus();  
let counter = 0;  
  
every(100, () => {  
  const data = [0, counter, 0x12, 0x34, 0x56, 0x78, 0x9A, 0xBC];  
  data[0] = e2eCrc('p01', data, { dataId: 0x123 }); // CRC in byte 0  
  can.send(0x123, data);  
  counter = (counter + 1) % 15;                      // 0..14  
});
```

Good to know

- **Timing.** Timers keep to their period without drifting, and a received frame is handled as soon as it arrives, without waiting for the trace display. Windows can still delay a timer by a millisecond or more now and then. Check the Cycle column in the trace.
- **A function that runs too long** (for example an endless loop) is stopped after 2 seconds, and so is the script.
- **Frames the script sends itself** are not received by that script. They show in the trace as **Tx**, and other interfaces on the same bus receive them.
- **CAN FD** frames need an interface ticked **CAN FD**. The Simulation channel takes them either way.