BG CALIBRATOR
-
REFERENCE MANUAL

Baldur Gíslason

January 22, 2019

Contents

1 Introduction

BG calibrator is a versatile user interface developed by Baldur Gíslason for use with all sorts of motorsport and industrial control electronics. The software is built on a modular approach and has few features that are specific to a single line of controllers. Therefore at the time of writing, to connect to a controller, a configuration file compatible with the controller's firmware is required. The configuration file describes the data format of the controller as well as which communications protocols can be used to connect. The configuration file opened does not have to match the config stored in the controller but it has to match the controller's firmware revision. The application will refuse to connect if firmware does not match, and a dialog will inform the user of the correct version. Configuration files are distributed with the controller firmware as well as downloadable on their own from the author's web site. `https://controls.is` If the configuration file required to connect to your controller can not be found, please contact the author. Configuration can be edited while connected to a controller or off-line without connection to a controller. Configuration files used by this application typically have the `.cud` file extension.

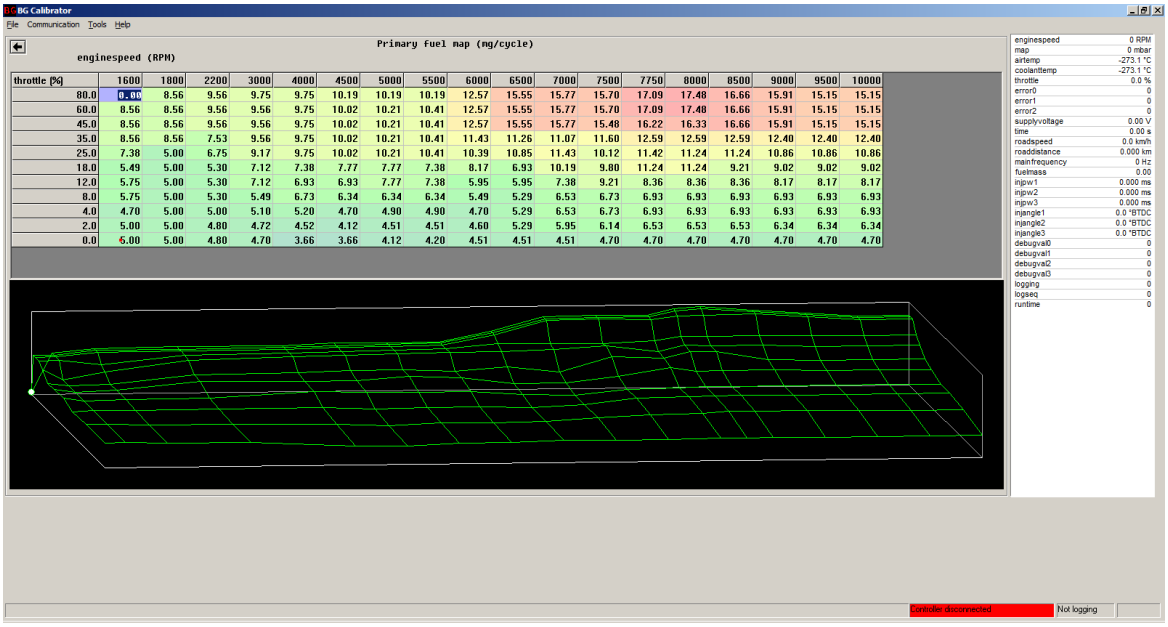


Figure 1.1: A typical view of the application

2 Getting started

2.1 Installation

2.1.1 System requirements

The application has been tested on 32- and 64 bit versions of Windows 7 SP1, Windows 8 and Windows 10. As of July 2017 Windows XP is unsupported and active development is done on Windows 10. The application requires minimal disk space and RAM. CPU power requirements vary depending on the use case, in general any computer made in the last decade should work.

2.1.2 Downloading

The application is distributed as a Windows Installer file (`.msi`). The latest version can be found at `https://controls.is/calibrator`

2.1.3 Installation and first run

Execute the msi file to install the application. Administrator privileges should not be necessary. Upon first run of the application, Windows firewall may prompt you to allow network access. This must be accepted if the control unit to be interfaced uses TCP/IP connectivity or if you would like the application to check for updates automatically. For USB or rs232 connected control units no network access is necessary.

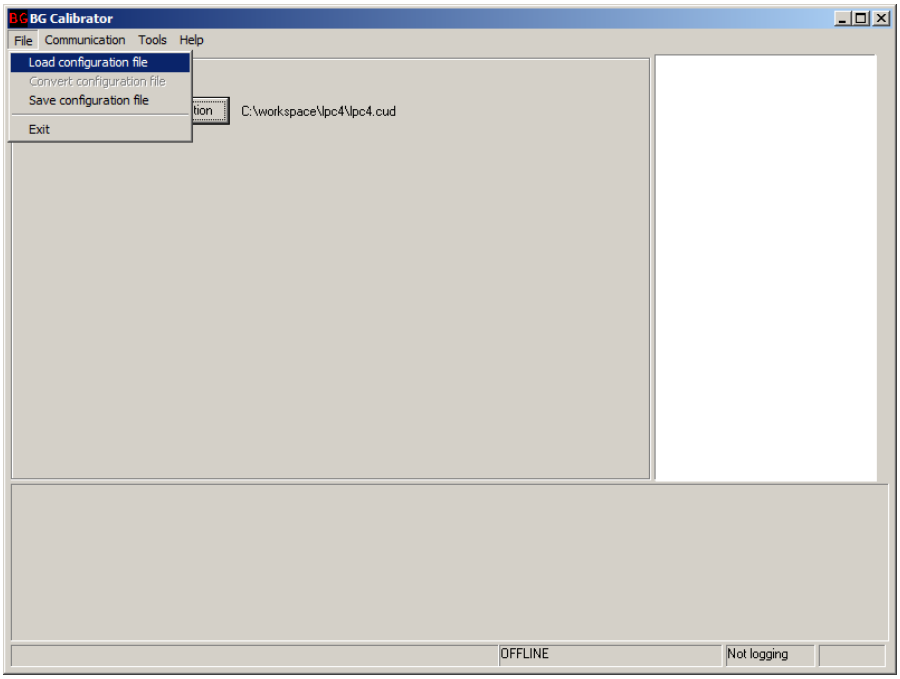


Figure 2.1: The initial screen

2.2 Opening a configuration file

To start with, open a configuration file. If you have done this before, the last known configuration can be re-opened using a button on the initial screen. Otherwise, select File -> Load configuration file from the menu bar at the top of the application dialog. Once the configuration has been opened you will be greeted by a dialog asking if you'd like to connect to a controller or work off-line. For certain communications protocols (eg. TCP/IP or RS232 based) this dialog will have a writable address field, where the IP address or COM port can be entered. The last known IP address or COM port is stored in the configuration file. If you choose to connect, the application will try connecting to the controller in the background and the status bar on the bottom of the screen will display the state of the controller connection. If you do not wish to connect at this time, you can

choose to do so later by selecting Communication -> Start from the menu bar.
If you do not have the correct configuration file for your controller, you should be able to acquire it from `https://controls.is/configs`. If you are unsure of which configuration you need, select Communication -> Scan for controllers from the menu bar. A dialog will pop up listing all connected controllers that may be supported by the software and their firmware versions and serial numbers.

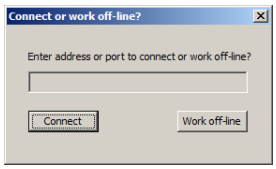


Figure 2.2: Connect dialog

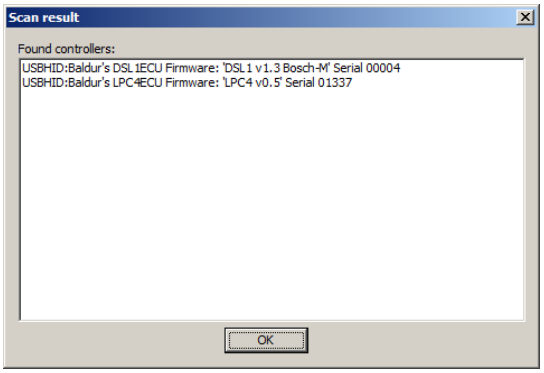


Figure 2.3: Controller scan results

2.3 Going on-line

Whenever a connection is opened with a controller, the application reads the controller's memory and compares its values to those found in the configuration file. If a difference is found, a dialog

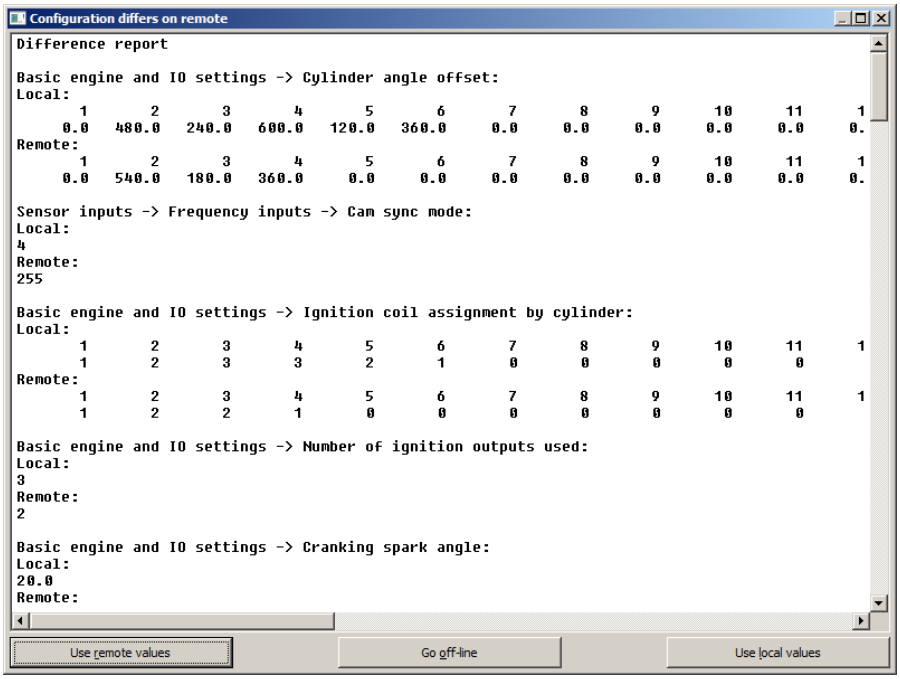


Figure 2.4: Difference dialog

pops up that illustrates those differences and presents the user with three options.

Use remote values The values found in the controller replace those found in the currently open configuration. Controller values are unmodified. This is the default option picked if the user pushes the return key.

Go off-line Cancels connection to the controller, no changes are made to either config.

Use local values Sends the values from the configuration file to the controller. A back-up of the controller's values is made in the same directory as the configuration file, named backup followed by a time stamp.

2.4 Editing configuration

Once the configuration has been opened, the main screen presents a tree view of the configurable parameters. Multi dimensional parameters will have a grid icon (▦) displayed on their left, and if they are function types their name will be followed by `f(x)` notation indicating that they are functions and if applicable, which real-time variables they are functions of. Single dimensional items will have a unit icon (1) displayed on their left and their name succeeded by their current value and unit of measure in square brackets.

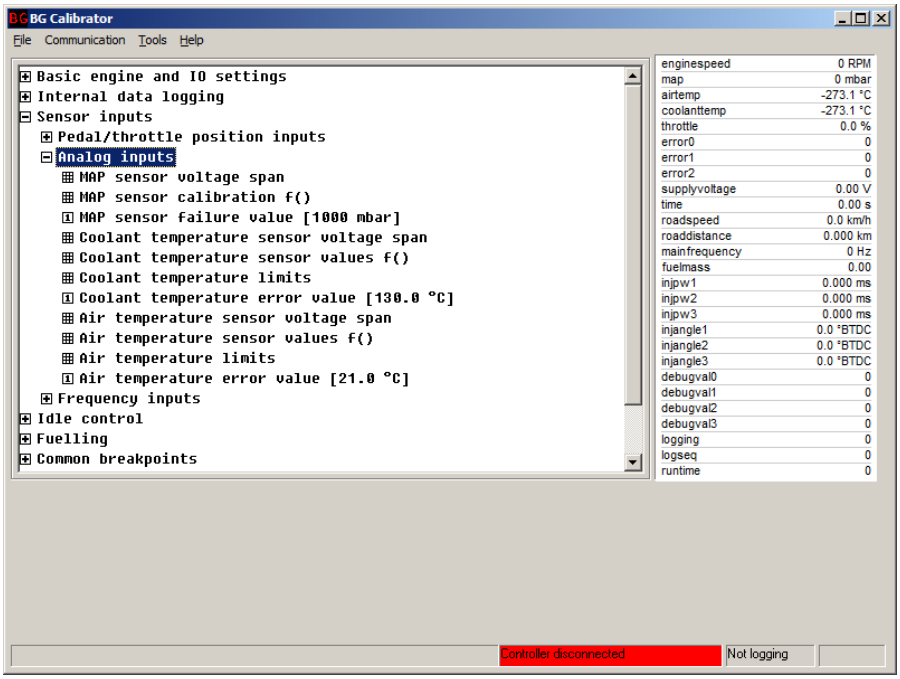


Figure 2.5: Tree view

The tree view is navigable using the mouse where double clicking will expand/collapse categories or or by using the arrow keys on the keyboard. Up/down arrows will navigate the list while left/right arrows will collapse/expand branches. The return key also serves to collapse/expand branches. Double clicking

or pushing return or right arrow on a config item will enter edit mode.
Edit mode is primarily keyboard operated but does accept mouse input for some purposes. Most of the ctrl-key functions listed are available under a right-click context menu in the editor as well. When in edit mode, the arrow keys can be used to navigate multidimensional items, holding the shift key enables selecting a range of data fields. Entering a number will apply that value to the selected fields upon press of the return key. Any operations such as increment/decrement/interpolate, etc are not sent to the connected controller until the return key is pressed or edit mode is left by means of the escape key or clicking the back button visible at the top left of the screen.
For enumerated values, pushing the return key drops down the list of options so it can be scrolled through. Pushing return again accepts the new selection.
At present, table axis can not be edited from within the table editor. The axis breakpoints must rather be defined and edited as separate items in the configuration.

Keys	Function
Esc	Leave edit mode
ctrl-Z	Leave edit mode and discard (undo) changes made
arrows	Move cursor
Shift + arrows	Select area
return	Commit changes
numbers	Enter new value for selected field(s)
A	Increase selected field(s) by the granularity of the item being edited
Z	Decrease selected fields(s) by the granularity
S	Increase by 10x the granularity
X	Decrease by 10x the granularity
D	Increase by 100x the granularity
C	Decrease by 100x the granularity
ctrl-A	Select all data fields for editing or copying
ctrl-S	Select all fields including axis for copying
ctrl-C	Copy selected fields
ctrl-D	Create keyboard shortcut to edit the current config item
ctrl-V	Paste at cursor
ctrl-M	Apply clipboard data as modifier (multiply/add/divide)
ctrl-J	Interpolate selection horizontally
ctrl-B	Interpolate selection vertically
ctrl-L	Transform selection (opens dialog)
ctrl-E	For breakpoints tied to variable, apply current value at cursor
ctrl-PgUp	Switch to previous config item in tree branch
ctrl-PgDn	Switch to next config item in tree branch

Figure 2.6: Keyboard shortcuts in edit mode

Keys	Function
F1	Switch bottom pane to context help. Invokes variable-specific context help in the
F2	Switch bottom pane to ticker view
F3	Switch bottom pane to dashboard
F12	Start/stop recording live data to file (data log)
Ctrl-F	Configuration variable search
Space bar	Mark event in data log if currently recording log to native log format

Figure 2.7: Global keyboard shortcuts

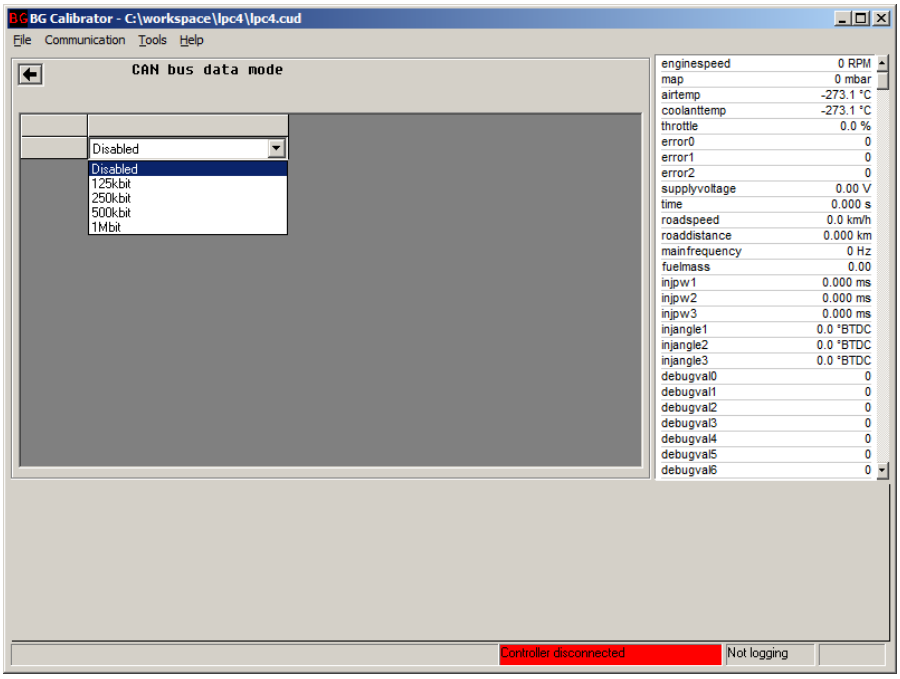


Figure 2.8: Example of an enumerated data type

3 Options

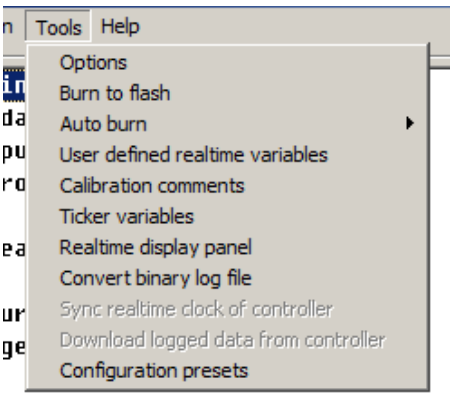


Figure 3.1: Tools menu

3.1 Options dialog

Under Tools -> Options from the menu bar at the top of the application, a dialog to configure the application can be found. Those settings are stored in the global application configuration and not as part of the opened configuration file. At present the following settings can be configured from the dialog.

Display frame rate Sets the redraw rate of live data displayed in the application. Has no effect on communication or logging rates.

Ticker duration Sets the time frame displayed in ticker view.

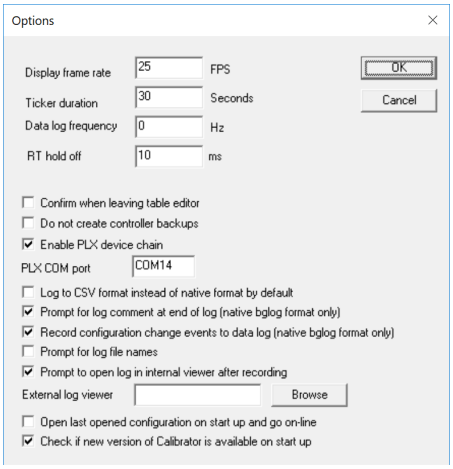


Figure 3.2: Application options dialog

Datalog frequency Can be used to limit logging rate if logs are to be taken over long periods or viewed with log viewers that are poorly behaved with large files. A zero means data is recorded as fast as it is received, any other value means the logger holds off on writing a new frame to the log for a duration of the inverse of that frequency.

Confirm when leaving editor If this option is selected, a dialog will ask you if you wish to make the changes you did to a configuration item permanent when leaving edit mode. If you then choose not to save the changes, the value of the item is reverted to its previous state, before edit mode was entered.

Enable PLX device chain The application supports logging additional data from sensors hooked directly to the PC using the PLX Devices iMFD protocol, so PLX SM-AFR and comparable devices can be logged and displayed. Once the correct COM port has been selected the devices will be automatically detected and created in the real time variables list with names that start with plx, such as `plxafr0` for the first wide band lambda sensor in the chain.

Log to CSV format instead of native format If selected, the default logging format will be comma separated values instead of the native format. Use this option if you prefer to use an external log viewer. Note that when logging directly to CSV format, not all logging features are available.

Prompt for log comment If enabled, a dialogue box will pop up when logging is ended allowing you to enter a description for the log session that just ended. If logging to CSV format this option has no effect.

Record configuration change events to data log If this option is selected, any config canges made during the log recording will be recorded in the log at the exact time they were made. This may prove useful to analyse the effect different settings have after the fact.

Prompt for log file names If selected, a file dialog will prompt the user for location and name to save new data log as. Otherwise the log file will be saved in the same directory as the currently open configuration file, with a name generated from the current date and time.

External log viewer The path to a log viewer application. If this field contains a value, the user will be prompted to open a log file with the viewer application when logging ends.

Open last opened configuration on start up and go on-line This option enables hands-off operation, useful if using the application for a dashboard display or something similar.

Check if new version of Calibrator is available on start up If this option is enabled, the application will connect to the web site on start up and check if the latest version available from the web site matches the currently installed version. If the version available on the web site is different, a button appears on the initial screen. The button disappears when a configuration is opened.

3.2 Burn to flash

This option manually forces the connected controller to write changes done to the configuration to non-volatile memory. The same effect can be achieved by clicking the controller memory state indicator in the bottom right corner of the application dialog. Most controllers will not immediately store changes to their configuration in non-volatile memory. This means if the controller gets switched off before the changes have been permanently saved, they can get lost, in which case the application will alert the user the next time the controller is connected, as the locally known configuration will not match that of the controller. If the user closes the application or stops communication before disconnecting or powering down the controller, saving data is taken care of automatically.

3.3 Auto burn

Auto burn selects how aggressively the application manages the process of automatically writing to the controller's non-volatile memory. Some controllers may disrupt normal operation during such writes so in those cases it makes sense to limit the number of writes done during operation. The options available are as follow.

Aggressive Writes all changes to non-volatile memory as soon as they're committed.

Lazy Writes changes to non-volatile memory only when leaving edit mode

Off Not really off. Changes are written to non-volatile memory only when necessary, such as when preparing to shut down the application or disconnect the controller, or when controller's physical memory arrangement prohibits storing all of the config in temporary memory, in which case some of it will need to be paged out to non-volatile memory to enable editing of other parts.

3.4 User defined realtime variables

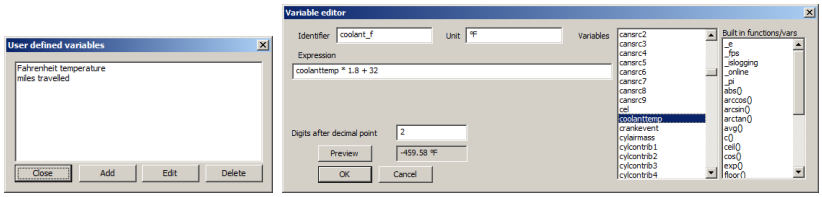


Figure 3.3: Examples of user defined variables calculated from existing variables

The application lets the user define real-time variables calculated from the values of any number of other real-time or configuration items. These are stored inside the configuration file and not in the app's global configuration. The basic syntax is detailed in chapter 6 but with the following additions that only apply to real time variables:

x Where x is the name of the real time variable, this returns the value of that variable.

c(x) Where x is the name of a unit configuration item, this returns the value of that item.

c(x, offset) Where x is the name of an array configuration item, this returns the value stored at a certain offset in the array. Offset can be a constant or a computational result.

clookup(x, value) Where x is the name of an array configuration item, this returns the index at which the value provided is found, with interpolation.
For example, where `fuelrpmbins` is an array of 8 values spanning from 1000 to 8000 evenly spaced, `clookup(fuelrpmbins, 1500)` returns a value of 0.5. `clookup(fuelrpmbins, enginespeed)` will return 0 if `enginespeed` is smaller than or equal to 1000, 7 if `enginespeed` is greater than or equal to 8000 and any number in the middle depending on the index on the curve.

_fps Returns the configured refresh rate of the application, useful for scripting.

_islogging Returns 1 if a data log is being recorded, 0 otherwise.

_online Returns 1 if controller is connected, 0 otherwise.

3.5 Calibration comments

A dialog that lets the user enter notes to store in the configuration file.

3.6 Ticker variables

A dialog for selecting which real-time variables are drawn in the ticker view.

3.7 Realtime display panel

This option launches a customisable dash dialog box. Not documented at present.

3.8 Convert binary log file

In case the controller the configuration file matches has the ability to record data on its own, this option is to open a raw log file and pass it through the log formatter of the application to convert it into a format compatible with most data viewer tools.

3.9 Sync realtime clock of controller

If the connected controller has a real time clock, this will sync that clock to the clock of the system running the application. This is a manually executed operation that must be run as

deemed necessary. Option will be greyed out if no controller is connected or if connected controller does not support this feature.

3.10 Download logged data from controller

If the connected controller has internal data recording, this will open up a dialog to manage the logs stored in the controller and download them from the controller. Greyed out if no controller is connected or controller does not support this feature.

3.11 Configuration presets

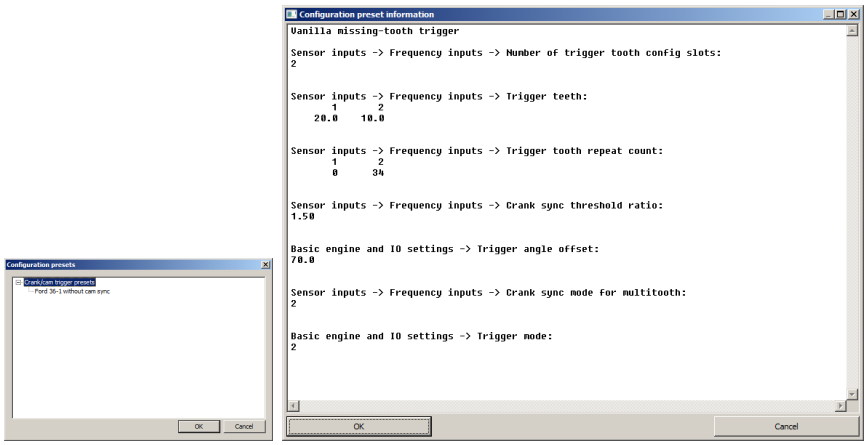


Figure 3.4: Configuration presets

If the opened configuration file contains any, this is where preset templates can be found for commonly used combinations, such as transfer functions for common sensors. Double clicking any of the present presets will bring up another dialog describing the preset, detailing which configuration options will be changed and what their new values will be after applying the preset. Clicking the OK button or pushing return in that dialog will apply the preset. Clicking cancel or pushing the escape key will close the dialog without applying the preset.

3.12 Capture event log

If a controller is connected that supports this feature, this option records a log of events going on at the inputs and outputs of the controller, namely crank/cam trigger inputs, wheel decoder state, ignition and injection outputs and more. The log is then displayed in a logic analyser like graphical display. If the crank trigger input is of impulse type (single edge) the crank pulses will be shown having varying height on the graph, the height indicating the period since the previous pulse, relative to the longest period shown on the screen. The sync channel if present has special formatting also, half sync is shown at half height, full sync is shown at full height and sync loss is shown in red. Clicking a sample will highlight it in an amber colour and the time stamp of that sample will be shown at the bottom of the screen. Then holding the ctrl key and clicking another sample will highlight that sample in a violet colour and the time stamp of the violet sample will be shown at the bottom of the screen along with the time delta between the samples.

3.13 View controller errors

If a controller is connected that supports this feature, this dialog will decode and display the error variables.

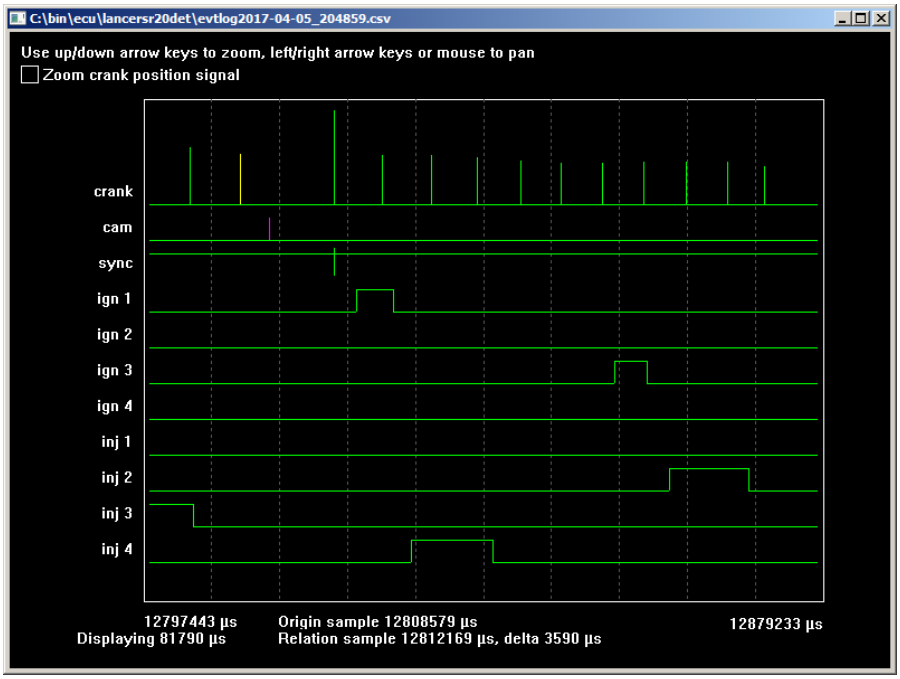


Figure 3.5: Event log view

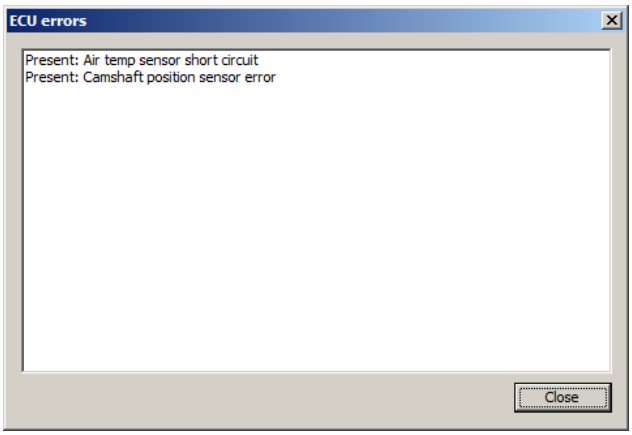


Figure 3.6: Error dialog

4 Configuration conversion

Whenever the firmware of a controller is updated in such a way that alters its internal data structure, an updated configuration file is required. The application can convert a configuration file to match the structure of another configuration file meant for the same type of controller and retain all of the existing settings that are compatible across versions, settings that exist on the new config but not on the old config will retain their value as found in the new config. User defined variables, calibration comments, dashboards and such settings that have to do with the application layout and not the controller will retain their settings from the old configuration file. If one wishes to get the new layout defaults, the best way is to use the converted file to transfer the settings to the controller and then connect to the controller using the default file for that firmware, reading the settings back from the controller to create a file that has the old settings but new dashboard layouts, etc.
The process for configuration conversion is as follows.

1. Open the old configuration file as usual, choose to work off-line.

2. Select File -> Convert configuration file from the menu bar. Pick a configuration file that matches the current version of the controller firmware.

3. Save the configuration file, it is ill advised to overwrite the old file.

5 Log viewer

The application provides a built in viewer for viewing log files. It can read most csv format log files, including but not limited to the ones recorded by this application. All log viewer settings, including chart templates and layout templates are stored in the application config and are independent of any control unit configuration files. The log viewer can be invoked from the logging menu in the application main window.
Once a data log file has been opened, the layout can be defined. The viewer uses tabs to present multiple view pages. Tabs may be selected by a mouse click or by pressing the number keys on the keyboard while holding the `Alt` key. Three tab types are currently available, one that displays charts as a function of time, one that displays a X/Y scatter plot showin values on the Y axis as a function of any other value and a histogram generator that displays averages of a variable as a function of one or two other variables. Each chart may be configured by a right click on the chart and selecting `Chart properties` from the context menu.
Following is a description of the menu bar at the top of the log viewer dialogue.

File :

Open reference file Opens a file for overlaying on the currently opened file. The variables in the reference file get their name suffixed :R and by holding the shift key while dragging across the charts the reference file can be repositioned in relation to the main file.

Append another log file at end of current data
Opens another file and the data from that file gets inserted beginning at 0.1 seconds after the last sample currently displayed.

Insert another file with compatible time stamps
Opens another file and the data from that file gets inserted wherever its timestamps place it in relation to the currently loaded data. Perfect for working with files generated during the same outing where the time stamps are from the same starting time.

Save session to file Save the currently displayed data to a BGLOG file for later review.

Export log to CSV format Save the currently displayed data to a CSV file for review in other software. Some details are lost, such as event markers and comments. CSV files also do not render data sources with varying sample rates optimally.

Close viewer Closes the viewer dialogue.

Edit :

Discard left of cursor Trims the loaded data, discarding everything before the position of the cursor (centered in the chart).

Discard right of cursor Trims the loaded data, discarding everything after the position of the cursor (centered in the chart).

Discard all but selection Trims the loaded data, discarding everything but the selected region.

Layout :

New tab Adds a new tab to the current layout from a list of possible blank tab types.

Load from tab template Adds a new tab to the current layout from a list of previously saved tab layouts.

Save this tab as template Saves the currently displayed tab for recall at a later time either from the internal tab template list or to a file.

Close tab Closes the currently displayed tab, but only if other tabs are present.

Close all tabs but current Close other tabs than the currently displayed one.

Undo close tab If a tab was closed inadvertently, this action will re-open it.

Log viewer settings Opens a dialogue box where colours and other aesthetic features of the log viewer may be configured.

Import layout from template file Opens a tab template (single tab) or workspace (multiple tabs) previously saved to a file.

Add chart If the tab currently displayed is a charts tab, this will add a new chart to the bottom of that tab and pop up a configuration dialogue for that chart.

Load chart from template If the tab currently displayed is a charts tab, this will add a new chart to the bottom of that tab according to a previously saved template.

Delete chart template This menu permits the deletion of previously saved chart templates from the internal template database.

Load workspace template Selecting an item from this menu will close all currently open tabs and open a set of tabs previously defined as a workspace template.

Save workspace template Saves the currently open tabs as a template for later use, either to the internal template database or to a file.

Delete workspace template This menu permits deletion of workspace templates from the internal template database.

Value display Toggles the value display pane on the right hand side of the screen for the currently displayed tab.

Show all variables If selected, the value display pane displays all of the variables found in the current log. Otherwise it displays only the variables relevant to the currently displayed tab.

Rename tab Lets you enter a new title for the currently displayed tab.

Resize tab items When selected the objects in the tab no longer respond to mouse or keyboard input but instead their boundaries may be dragged around. Select this menu again to finish resizing.

Tools :

User defined variables Opens a dialogue that lets you define your own channels as math functions of other channels present in the log.

Decode error variables If a configuration file is open in the main window and the log file currently displayed is generated by the same controller firmware as the configuration originates from, this menu lets you decode bit mask variables that describe diagnostic trouble codes.

Recompute user defined variables If any changes have been made to the log or one or more user defined variables, this menu can be used to re-run the computation of the user defined variables.

Log comments Displays comments associated with the currently displayed log file if any, and lets you enter new comments that will be retained if the file is then saved as a BGLOG file.

Copy tab screenshot to clipboard Copies a screenshot of the currently displayed tab to the Windows clipboard, which can then conveniently be pasted to e-mail or instant messenger for review.

Save tab screenshot to file Saves a screenshot of the currently displayed tab to a PNG image file.

5.1 Charts

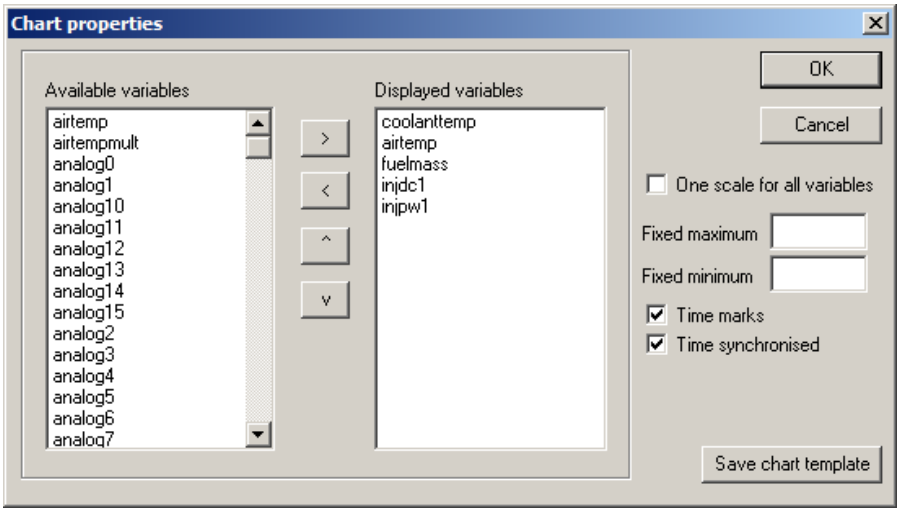


Figure 5.1: Log chart configuration

The chart properties dialog has a number of features, the most notable being the ability to select which log channels are displayed. Other notable features:

One scale for all variables If this option is selected, the chart vertical axis will be scaled in such a way that all variables are presented on the same scale, as decided by the greatest and smallest values of all of the variables on the chart. This is useful if all of the variables displayed are of the same type (injector pulse widths for different cylinders, exhaust temperatures of different cylinders, etc).

Fixed maximum If a value is set, this option fixes the upper limit of the scale for all variables. May be set independently of fixed minimum.

Fixed minimum If a value is set, this option fixes the lower limit of the scale for all variables. May be set independently of fixed maximum.

Time marks Draws a vertical line across the chart at time intervals of whole seconds, every ten seconds or whole minutes, depending on zoom level.

Time synchronised If set, this chart is synchronised to the other charts in centre time and zoom level. If not set this chart can be panned and zoomed independently of other charts.

Save chart template This button lets you save the displayed chart configuration as a template to be loaded when creating a new chart at a later time.

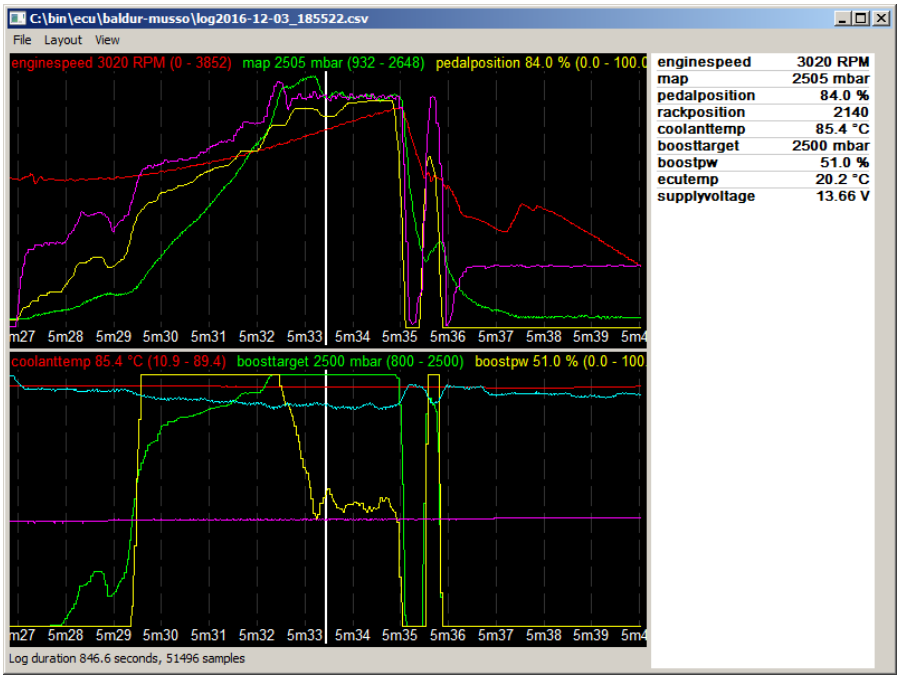


Figure 5.2: Log viewer dialog frame

The charts may be manipulated in the following ways.

Left/Right arrow keys The left or right arrow keys serve to pan the log, each press pans by one fiftieth of the displayed

time scale, unless the `Ctrl` key is held, then the panning step size is one log sample. Holding the `Shift` key enables selecting a time span of the log using the arrow keys.

Up/Down arrow keys The up or down arrow keys serve to increase or decrease the zoom level. They halve or double the displayed time span respectively.

Mouse left click By clicking anywhere in the chart, the clicked location is centered.

Mouse left click and drag Clicking and dragging selects a time span in the log.

Mouse left click and drag with CTRL pressed Pans the log along the time axis, also pans vertically when using discrete plots.

Mouse middle click and drag Pans same as left click+CTRL.

Mouse left click and drag with SHIFT pressed Pans the reference log along the time axis if one is loaded.

Mouse right click A right click opens a context menu, where the chart may be configured or deleted, and if a time span is currently selected the zoom may be set to the selected span.

Mouse scroll Same effect as up/down arrow keys.

5.2 X/Y plot

In the X/Y plot mode, one variable can be selected for the X axis and up to 16 variables can be selected for the Y axis. Configuration is otherwise mostly identical to the time based charts. The X/Y plot page has one time based chart and the purpose of that is to be able to limit the time range displayed on the plot. A selection on the time based chart lets the plot filter out all samples outside of the selected time frame.

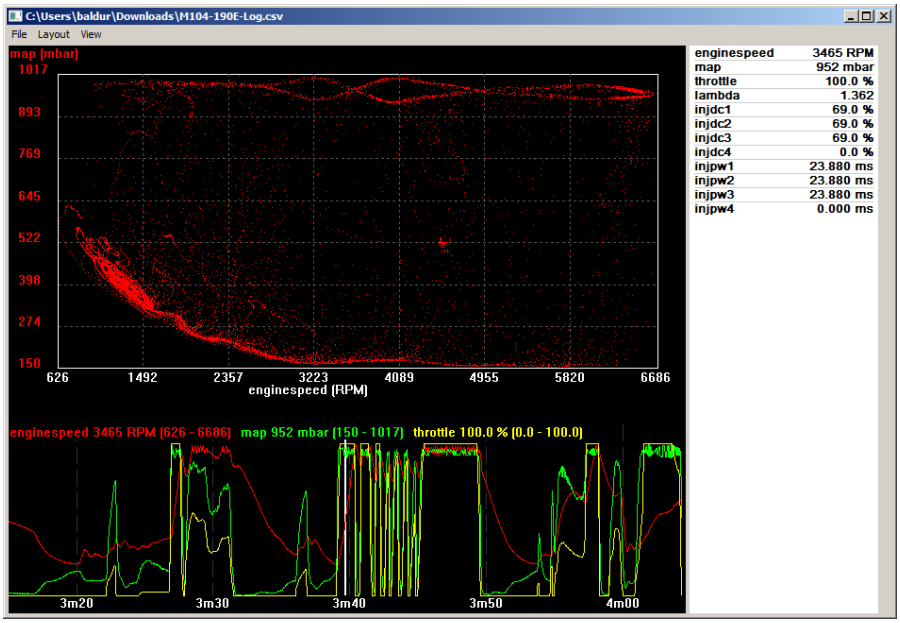


Figure 5.3: X/Y plot layout

5.3 Histogram

In the histogram mode, weighted average values over the entire log or a selected period are shown for one variable in relation to two other variables. The hits are weighed by how close to the table cell the sample hits, so a sample that hits the dead centre between four cells on the table affects each one with a weight of 0.25 while a sample that hits a cell directly affects only that cell by a weight of 1. The cells are coloured to show the hit rate, with the cells that represent the greatest number of samples coloured bright green.
When configuring the histogram, the X and Y axis can be configured either by entering the number of cells to break the axis into for automated cell labelling or by entering the values for all of the cells separated by spaces or tabs to manually define the cells. Cells with an unacceptably small number of hits may also be hidden from view, cells with no hits at all will not have any

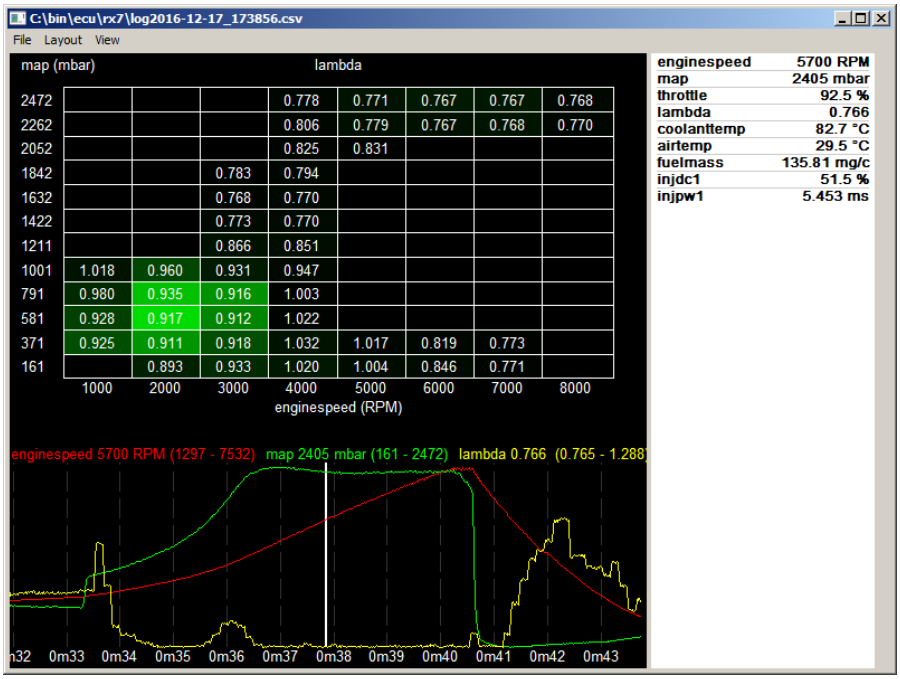


Figure 5.4: Histogram layout

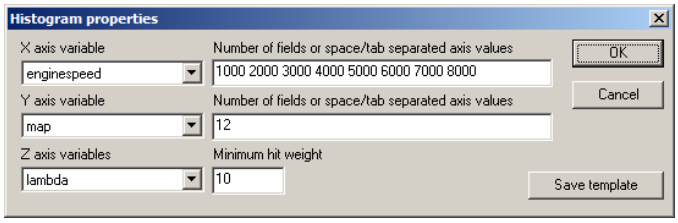


Figure 5.5: Histogram properties

value displayed regardless of filter setting. The hit values may be displayed by right clicking the table and selecting the `Show hit count` option.

5.4 Interval report

In the interval report mode, weighted average values of selected channels are gathered as a function of a single channel at a configured interval. Useful for finding trends in multiple data channels as a function of lap number or engine speed for example. The default is to show the data as a table and the peak value of each channel analysed is underlined. By right clicking the interval report the display may be switched to graphical mode. As with the histogram, the values are collected as a weighted average, with values that fall half way between cells in the input axis variable evaluated at half weight, but in the case of the interval report any sample only contributes to one cell. Like the histogram and scatter plots, the interval report offers the ability to use a variable to selectively filter the data to accept only samples from time periods when a certain variable has a non-zero value.
If multiple logs are opened at once and all log windows have the same layout, it is possible to overlay and compare data from multiple log files. You can synchronise the layouts by selecting `Sync all other windows to this layout` from the layout menu in the log viewer window. To bring up the reference data, right click the interval report and select the files you want to compare with under the `Toggle reference data from other logs` menu. Optionally, you can select to have the software highlight the highest and lowest values in each row. If any of the reference logs have a comment entered, the first line of the comment will be displayed along with the file name on the top of the screen. If the tab layout of any of the opened logs is changed so that the current window is not identical to the referenced window, or the interval report configuration is not identical the compared data will look strange until the windows are all synchronised to the same layout again.

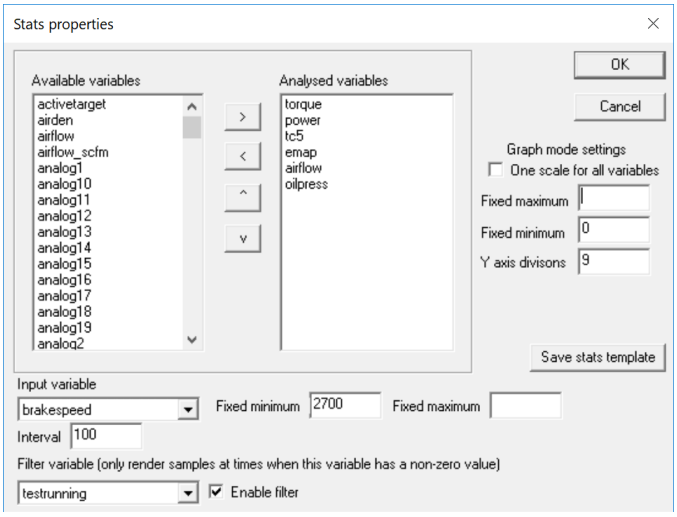
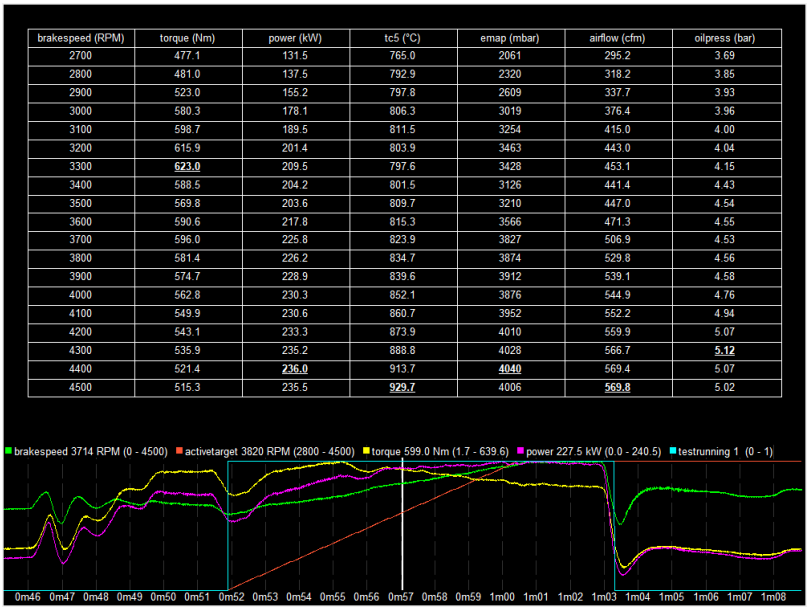


Figure 5.6: Property dialog for interval report



brakespeed (RPM)	torque (Nm)	power (kW)	tc5 (°C)	emap (mbar)	airflow (cfm)	oilpress (bar)
2700	477.1	131.5	765.0	2061	295.2	3.69
2800	481.0	137.5	792.9	2320	318.2	3.85
2900	523.0	155.2	797.8	2609	337.7	3.93
3000	580.3	178.1	806.3	3019	376.4	3.96
3100	598.7	189.5	811.5	3254	415.0	4.00
3200	615.9	201.4	803.9	3463	443.0	4.04
3300	**623.0**	209.5	797.6	3428	453.1	4.15
3400	588.5	204.2	801.5	3126	441.4	4.43
3500	569.8	203.6	809.7	3210	447.0	4.54
3600	590.6	217.8	815.3	3566	471.3	4.55
3700	596.0	225.8	823.9	3827	506.9	4.53
3800	581.4	226.2	834.7	3874	529.8	4.56
3900	574.7	228.9	839.6	3912	539.1	4.58
4000	562.8	230.3	852.1	3876	544.9	4.76
4100	549.9	230.6	860.7	3952	552.2	4.94
4200	543.1	233.3	873.9	4010	559.9	5.07
4300	535.9	235.2	888.8	4028	566.7	**5.12**
4400	521.4	**236.0**	913.7	**4040**	569.4	5.07
4500	515.3	235.5	**929.7**	4006	**569.8**	5.02

Figure 5.7: Interval report of a dyno test

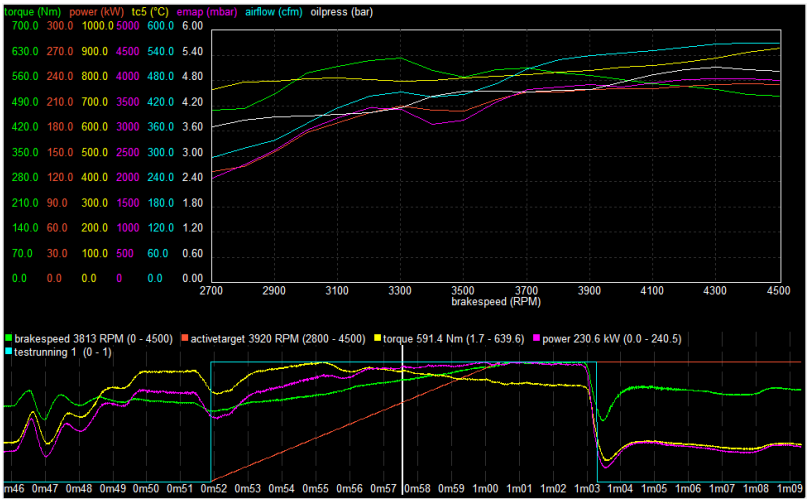


Figure 5.8: Interval report of a dyno test in graphical mode

1: C:\bin\dyno\log2019-01-01_153410.bglog : water meth
2: C:\bin\dyno\log2019-01-01_154251.bglog

brakespeed (RPM)	horsepower (hp)	1:horsepower (hp)	2:horsepower (hp)
3000	242.6		245.2
3100	250.3	253.8	249.6
3200	257.3	259.6	257.6
3300	263.3	265.5	262.5
3400	265.2	267.9	265.8
3500	269.8	271.0	273.3
3600	279.4	278.8	284.3
3700	280.2	286.8	289.1
3800	286.4	291.9	296.9
3900	292.9	298.5	301.5
4000	292.4	304.2	306.2
4100	296.4	305.6	312.9
4200	297.1	309.1	316.4
4300	299.9	314.2	312.1
4400	304.1	320.7	314.1
4500	307.9	325.0	320.0

Figure 5.9: Interval report comparing multiple logs, with the lowest values highlighted in red and the highest values highlighted in green. These colours can be changed in the log viewer settings.

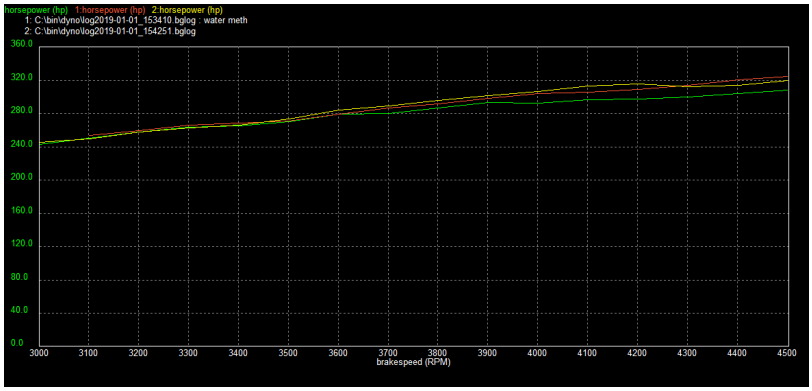


Figure 5.10: Interval report comparing multiple logs in graph mode.

5.5 Reference log overlay

The application allows overlaying a second log file's data on top of the loaded log file. To do this, select `File -> Open reference file` from the menu bar at the top of the screen. Variables from the reference log will be listed with the suffix `:R` in their name. To line up the times on the reference log and the main log, add a reference channel to one of the chart views and then click the chart with the left mouse button while holding the `Shift` key and drag until the logs line up.
Two shortcuts are provided in the right click context menu in the chart view to speed up bringing up the reference channels.
The user defined variables are also computed for the reference log but in order to refresh them after changing or adding variables, the reference log must be reloaded.
In case you wish to compare different parts of the same log, the same file can be loaded for main log and reference.

5.6 User defined log variables

The application lets the user define log variables calculated from the values of any number of other log variables or if a configuration is loaded, items from the loaded configuration may be referenced as well. These are stored inside the app's global configuration.

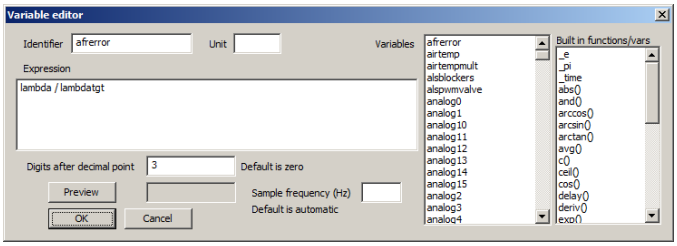


Figure 5.11: Dialog for configuration of user defined variables

The basic syntax is detailed in chapter 6 but with the following additions that only apply to log variables:

_time Returns the time stamp of the sample being computed, starting at zero.

delay(x,d) Returns the value of x delayed by the time specified by d. d can be negative to advance instead of applying a delay.

deriv(x,t) Returns the derivative of x at the current point in time, where t specifies the sample time span. If only one argument is given, then the last sample is used for reference. For example: `deriv(enginespeed, 0.5)` returns the change in `enginespeed` by comparing a sample a quarter of a second prior to a sample a quarter of a second ahead. If t is negative, then instead of looking ahead to find the current derivative, the derivative is taken as the difference between the current sample and a sample prior in time to the current sample.
The most accurate results are had if the derivative function is applied directly on a log variable instead of taking the derivative of the result of some expression due to the way time stamps are handled with multiple channels.

lowpass(x,d) Applies an exponential decay filter to the value of x, where d specifies the amount of filtering done. d has a valid range of 0 to 1 where 0 is no filtering at all and 1 is absolute filtering, no data gets through.
For example: `lowpass(enginespeed, 0.9)` returns the value of enginespeed filtered in such a way that only a tenth of the ampliitude of the change each sample represents gets passed through.

sum(x, cond) Accumulates the value of x across the entire duration of the log, useful for the purposes of integration. Takes an optional second argument, which zeroes the value of the accumulator if its value is zero.

c(x) Where x is the name of a unit configuration item, this returns the value of that item.

c(x, offset) Where x is the name of an array configuration item, this returns the value stored at a certain offset in the array. Offset can be a constant or a computational result.

overflow(x,max,min) An overflow filter for variables that have a limited range, used for example if you have a counter that counts from 0-255 and starts over, and you want to keep track of the accumulated value which is much larger than 255. Takes at least one, at max three arguments, first being the input value and the second optional argument is the value at which the input overflows, that is one greater than the maximum count (the value that the counter would have reached on overflow, if it hadn't overflowed), defaults to 256 if not specified. The third argument is the expected starting value after overflow, defaults to zero if not specified.

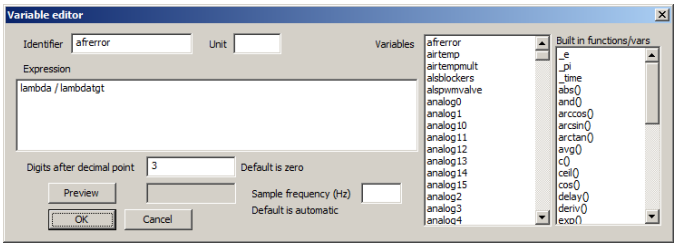
latch(x,reset) A function that takes one argument and simply remembers the first value passed to it and remembers it, always returns that value on every successive run. An optional second argument may be passed, which will reset the output value to the current value of the first argument if the second argument's value is non-zero

5.7 User defined log variables examples

Being able to create custom variables via mathematical expressions enables endless possibilities in quick and efficient interpretation of log data. Below are some examples of the possibilities.

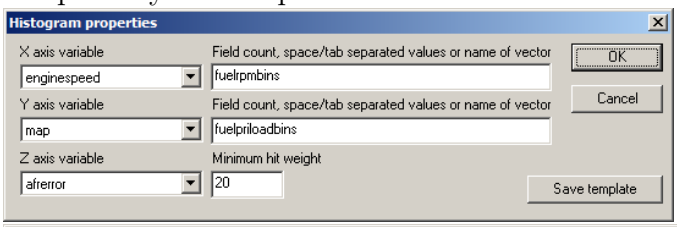
5.7.1 Auto tuning of fuel table

In this example used on an LPC4 ECU, a custom variable is created that shows how far off target the air:fuel ratio is in a log: `lambda / lambdatgt`



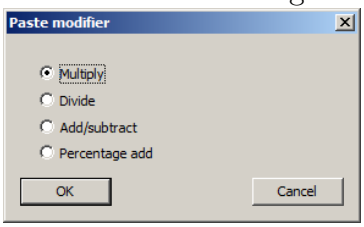
Alternative: `delay(lambda, -0.1) / lambdatgt` if the lambda sensor has slow response the signal can be advanced, in this case by 0.1 seconds.

Next step, use the histogram view to plot a table on the axis of the primary fuel map.



Now, select all fields in the histogram generated table and copy to clipboard. Go edit the primary fuel map and use the modifier

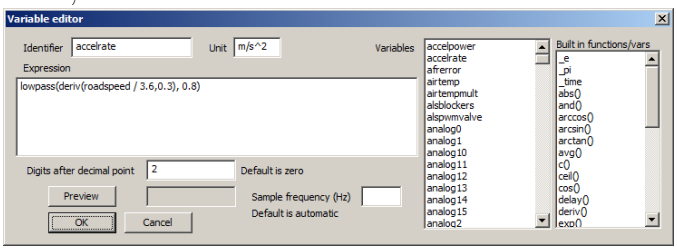
paste function (ctrl-M) to multiply the values of the table by the values from the histogram view.



5.7.2 Measuring acceleration rate, estimating power output

In this example used on an LPC4 ECU but applicable to anything, the rate of acceleration is calculated as the derivative of road speed. Since the rate of acceleration is a function of thrust (aka torque after gearing) and vehicle weight it can be used to determine how much of the power delivered to the wheels goes toward accelerating the vehicle. The same function can as well determine the power used to slow the vehicle down, which if the vehicle is coasting in neutral on a flat road equates the aerodynamic and rolling drag in one figure. If the drag is determined as a function of speed it can then be added on top of the acceleration power to come up with a true power figure and a power curve that is the correct shape.

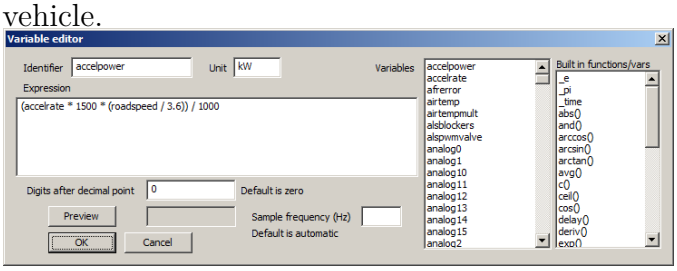
First, create a variable that takes the derivative of the speed.




```
accelrate = lowpass(deriv(roadspeed / 3.6, 0.3), 0.8)
```

In this example, road speed is divided by 3.6 to convert it from kilometres per hour into metres per second, then a derivative is taken over 0.3 seconds and the result filtered by 80% as the road speed signal has some noise to it.
The output of this expression is the acceleration in metres per

second per second (metres per second squared, remember 1 gravity is said to be 9.82 in the same units).
To convert the rate of acceleration to a power figure, it needs to be multiplied by the speed again as well as the mass of the vehicle.




```
accelpower = (accelrate * 1500 * (roadspeed / 3.6)) /
1000
```

Shortened:

```
accelpower = (accelrate * 1500 * roadspeed) / 3600
```

In this case, 1500kg is the mass of the vehicle and road speed is again divided by 3.6 to convert from km/h to m/s. The result is then divided by 1000 to convert from watts to kilowatts. Remember, a watt is one newton metre per second. The rate of acceleration multiplied by the mass is the force pushing the vehicle forward and the road speed is the rate at which the work is done.


Baldur's data viewer - C:\bin\ecu\accord2007\log2017-02-07_223652.csv
File Layout View Tools
enginespeed 3336 RPM (1622 - 4593) map 1636 mbar (159 - 1762) throttle 90.9 % (3.1 - 98.3) lambda 0.822 (0.770 - 1.292)
accelrate 2.22 m/s^2 (-1.43 - 2.45) accelpower 72 kW (-50 - 101)
airtemp 19.0 °C (18.5 - 20.5) lambdatgt 0.80 (0.80 - 1.02) afrerror 1.027 (0.755 - 1.267) roadspeed 77.4 km/h (41.7 - 107.0)
0m04 0m05 0m06 0m07 0m08 0m09 0m10 0m11 0m12 0m13 0m14 0m15
Log duration 23.3 seconds, 1055 samples

6 Mathematical expressions

Starting in versions released after mid February 2017, mathematical expression support has been greatly expanded by the development of a new math expression parser system that is not based on an external library and thus integrates better with every aspect of the application. The syntax follows most established concepts in in-line mathematics, with named variables and function calls on the form of `f(x)` with commas separating multiple arguments where applicable. The order of operations is alterable by parentheses. As this is a new development any user feedback would be appreciated, namely if any bugs are discovered in the calculation or requests for operators enabling certain type of computation.

6.1 Built in operators and variables

\+ Adding operator.

\- Subtraction operator, may also be used to invert the sign of a number by placing inside parentheses, for example `2 * (-2)` has a result of -4.

* Multiplication operator.

/ Division operator.

^ Power operator, raises the left number to the power of the right number. For example `4 ^ 2` returns 16.

% Integer remainder operator, returns the remainder of an integer division. For example `12 % 5` return 2.

= Equals operator, returns 1 if numbers on both sides are equal, 0 otherwise.

> Greater than operator, returns 1 if the number on the left has a greater value than the number on the right.

< Less than operator, returns 1 if the number on the right has a greater value than the number on the right.

& Bitwise integer AND operator.

| Bitwise integer OR operator.

_e Euler's constant, e

_pi Pi, π

min(...) Returns the lowest value from a list of up to 16 arguments. Example: `min(4,1,7)` returns 1.

max(...) Returns the greatest value from a list of up to 16 arguments. Example: `max(4,1,7)` returns 7.

avg(...) Returns the average value of the list of up to 16 arguments. Example: `avg(3,1,8)` returns 4.

sqrt(x) Returns the square root of the argument. Example: `sqrt(4)` returns 2.

log(x) Returns the base 10 logarithm of the argument. Example: `log(100)` returns 2.

ln(x) Returns the natural logarithm of the argument. Example: `ln(_e)` returns 1.

abs(x) Returns the absolute value of the argument, that is, strips the negative sign away if necessary.

pos(x) Limits the value to zero and above. Returns the value of the argument if the argument is positive, otherwise zero.

neg(x) Limits the value to zero and below. Returns the value of the argument if the argument is negative, otherwise zero.

exp(x) Exponent operator, returns the Euler constant to the power of the argument.

ceil(x) Rounds the argument up to the next integer value.

floor(x) Rounds the argument down to the next integer value.

sin(x), cos(x), tan(x) Trigonometric functions, taking arguments on radian form.

arcsin, arccos, arctan(x) Inverse trigonometric functions in radian form.

if(x,y,) Ternary operator. Takes two or three arguments. If the first argument has a value other than zero, it returns the value of the second argument. Otherwise it returns zero or if specified, the value of the third argument.
Example: `if(tps > 90, rpm)` returns the value of `rpm` if the value of `tps` is greater than 90, otherwise zero.
`if(roadspeed > 10, rpm / roadspeed, -1)` Returns the ratio of `rpm` divided by `roadspeed` if `roadspeed` has a value greater than 10, returns -1 otherwise.

not(x) Bitwise integer NOT operator, returns the inverse of all bits in argument.

xor(x,p) Bitwise integer XOR operator.

lshift(x,s) Bitwise left shift operator, treats x as integer and shifts bits left by s.

rshift(x,s) Bitwise left shift operator, treats x as integer and shifts bits right by s.

and(...) Logical AND operator, evaluates up to 16 arguments and returns 0 once it finds an argument that has a zero value. Further arguments are not evaluated. Returns 1 if all arguments have non-zero value.

or(...) Logical OR operator, evaluates up to 16 arguments and returns 1 once it finds an argument that has a non-zero

value. Further arguments are not evaluated. Returns 0 if all arguments have a zero value.

lookup(c,d,x) Lookup on a curve with two or more points of data with linear interpolation. c specifies the breakpoints. d specifies the data points. x is the lookup value. The breakpoints and data points are constants, expressed as comma separated lists of numbers enclosed in quotation marks.
This function is highly useful to convert the value of a non-linear sensor to real units or express any other non-linear function that is derived from measured data.
Example: `lookup("1,2,3,4","100, 200, 300, 400", x)` returns a value of 100 if x is equal to or smaller than 1, a value of 400 if x is equal to or greater than 4, a value of 200 if x is equal to 2 and a value of 350 if x is equal to 3.5.

7 Scripting

Many aspects of the application are scriptable by advanced users. The scripting support was developed for dynamometer operation but is applicable to other applications as well. The scripts are structured as nestable json arrays with each instruction being an array of which the first (and possibly only) item denotes the name of the instruction to be executed. Script files may include Javascript style comments but comments in the configuration files are stripped out when the file is saved. The scripts run asynchronously so callbacks are used for blocking operations. Loops are run at the refresh rate of the application as configured in the settings dialog. If application is set to run at 50 frames per second, the script loops will run at 50 frames per second. Many of the script instructions return a boolean value, and if they do return false, execution of the branch the instruction is in will end. If an instruction is to be allowed failure without affecting execution of following instructions, wrap the instruction in an if statement or try statement.
Some instructions accept expressions as arguments, the expression format is the same as described in the last chapter and in the user defined variables chapter with the folllowing added operators:

widgetval(wid) Returns the value of a number input widget in a script dialogue, where wid is the widget identifier.

__rows Returns the number of rows in the table currently being edited. Behaviour not defined when not in edit mode.

__cols Returns the number of columns in the table currently being edited. Behaviour not defined when not in edit mode.

__cursorrow Returns the row the edit cursor is currently at in edit mode, with row 0 being the bottom row.

__cursorcol Returns the column the edit cursor is currently at in edit mode, with column 0 being the left most column.

7.1 Script invocation

There are a number of ways to invoke scripts. This section covers where scripts may be defined and invoked.

7.1.1 Keyboard shortcuts

In the configuration file, under the key `"keyshortcuts"` scripts can be tied to function keys F1 through F12, although keys F1-F3 and F12 have default functions in the application they can be overridden by scripting. Example:

```
1   "keyshortcuts": {
2     "F5": [ "edit", "mainfuelmap" ],
3     "F6": [ "edit", "mainignmap" ]
4   }
```

7.1.2 Library

In the configuration file, in the definition section under the key `"scripts"` scripts can be defined by name to be invoked by the `library` instruction or if they have a description attached to them, they can be found under the `Tools` menu in a submenu named `Script library`. In the main configuration body, under the key `"userscripts"` library scripts can also be defined. Example:

```
1 "scripts": {
2   "canscan_interactive": {
3     "description": "Interactive CAN scan",
4     "script": [
5       [ "rpc", "startscan" ],
6       [ "onexit", [ [ "rpc", "endscan" ],
```

```
7           [ "alert", "Scan ended at address \%s", "canscanaddr" ] ]
8         ],
9         [ "statusdialog" ],
10        [ "sleep", 200,
11          [ "while", "rt(\"canscan_running\") > 0",
12            [ "status", "Transmitting on address \%s", "canscanaddr" ]
13          ]
14        ]
15      ]
16    }
17  }
```

7.1.3 Events

Some events can be tied to scripts in the configuration file.

Configuration open If the configuration contains a key named `"onload"` that will be invoked when configuration is opened.

Configuration item edit If the configuration item being edited has a key named `"onopen"` that will be invoked when that item is opened for editing.

Configuration item value change After a new value for a configuration item is sent to the connected controller, a script specified by the `"onchange"` key is invoked if it exists. This happens both in edit mode and when sending local differences over to controller when going on-line.

7.2 Script instruction set

7.2.1 set

Sets the value of a config variable and sends the value to the connected controller.
Takes 2 arguments, variable name and an expression of the value to set. Expression follows same format as user defined RT vars unless the config variable is an enumerator, in which case the string value is used.
Returns false if there is an error.

Examples:
Set targetspeed to 3000:
```
[ "set", "targetspeed", 3000 ]
```
Increment targetspeed by 100:
```
[ "set", "targetspeed", "c(\"targetspeed\") + 100" ]
```

7.2.2 runscript

Loads script from file and runs it.
Takes one argument, the full path to the script file.
Returns false if there is an error running the script or the file is not found.
Example:
```
[ "runscript", "c:\dyno\cannedcycle.calscript" ]
```

7.2.3 hold

Waits for condition to become true, then starts a timer that delays an action until after condition has held truth for a given amount of time. If the condition becomes false, the timer is reset.
Takes 3 arguments, condition string in expression format, the hold time in milliseconds and a callback to run when timer runs out.
Example:
```
[ "hold", "rpm > 2000 && rpm < 2500", 3000, [ "alert",
"Success"] ]
```

7.2.4 sleep

Delays execution of a script for a given amount of time
Takes two arguments, the delay time in milliseconds and the callback to run when the timer is out.
Example:
```
[ "sleep", 2000, [ "alert", "Two seconds later" ] ]
```

7.2.5 while

An asynchronous while loop.
Takes 2-3 arguments, an expressional condition, a callback to run while the condition is true and optionally, a callback that runs once condition becomes false and the loop exits.
Example:

```
[ "while", "enginespeed < 2000", [ "set", "throttletarget",
"c(\"throttletarget\") + 1" ] ]
```

Example with exit callback:

```
[ "while", "enginespeed < 2000", [ "set", "throttletarget",
"c(\"throttletarget\") + 1" ], [ "alert", "Ready to
test" ] ]
```

7.2.6 if

An if/else condition.
Takes 2-3 arguments, an expression, a callback to run if true and optionally a callback to run if condition is false.
In place of an expression, a script that returns a value may be used. Examples:

```
[ "if", "enginespeed = 0", [ "alert", "Engine is not
running" ] ]
[ "if", "enginespeed = 0", [ "alert", "Engine is not
running" ], [ "alert", "Engine is running" ] ]
[ "if", [ "confirm", "Party?" ], [ "alert", "The user
likes to party" ] ]
```

7.2.7 try

Runs a subscript but returns true regardless of return value of subscript. Optionally accepts another script argument to be run if the first one fails.
Used if parts of the script are allowed to fail without ending script execution.
Examples:

```
[ "try", [ "burn" ] ]
```

```
[ "try", [ "burn" ], [ "alert", "Did not save to flash"
] ]
```

7.2.8 error

Aborts script execution, displays error dialog. Takes 1-9 arguments, the first being an error message and if the error message contains a format string, the rest of the arguments can be names of RT variables or config variables whose values to display.
Examples:

```
[ "error", "Engine has stalled" ]
[ "error", "Engine overheating:  %s C", "coolanttemperature"
]
```

7.2.9 return

Aborts execution of a single script thread by returning a false value.

7.2.10 log

Prompts user for file name and starts recording data log
Takes no parameters
Example: `[ "log" ]`

7.2.11 endlog

If data is being recorded, the recording is ended.
Takes no parameters

7.2.12 debug

Prints a message to the debug console.
Takes 1-12 arguments, the first being a format string and the rest being optional variable names (rt or config vars)
Examples:

```
[ "debug", "Everything is alright" ]
[ "debug", "Speed is %s RPM", "enginespeed" ]
```

7.2.13 logevent

Prints a message to the log being recorded, if recording a log in the native log format. Otherwise identical to the debug function in operation.
Takes 1-12 arguments, the first being a format string and the rest being optional variable names (rt or config vars)
Examples:

```
[ "logevent", "Started acceleration" ]
[ "logevent", "Set hold speed to %s", "holdspeed" ]
```

7.2.14 clipboardprint

Formats a text string and copies to clipboard. Works identical to other text formatting functions. Takes 1-12 arguments.

7.2.15 alert

Displays an informative dialog box. Pauses script progression until dialog is closed.
Takes 1-9 arguments, the first being a format string.
Example:

```
[ "alert", "Engine temperature is %s and oil temp is
%s", "coolanttemperature", "oiltemperature" ]
```

7.2.16 confirm

Displays a dialog, pausing script progression and allowing the user to decide whether to continue script execution or not.
Takes 1-9 arguments, the first being a format string.
Returns false if the user clicks the Cancel button.
Example:

```
[ "confirm", "Oil temperature is only %s C, do you
wish to proceed with test?", "oiltemperature" ]
```

7.2.17 createrтvar

Declares user defined RT variables which can from then on be used in the script as well as other scripts, displayed on a gauge

or recorded to a data log
Takes one argument, an array of json definitions as used in configuration file. Variables may be redeclared.
Example:

```
[ "createtvar", [
    { "id":"speederror",
       "expression": "rt(\"enginespeed\") - c(\"speedtarget\")"
    }
  ]
],
[ "alert", "Speed is \%s from target", "speederror" ]
```

7.2.18 declare

Declares a variable that is global in script context and whose value can only be updated by running the declare instruction again. Takes two or three arguments. First argument is the variable name, second is an expression and third which is optional sets the number of digits after the decimal point when the variable is printed using any of the text formatting instructions but has no effect on the precision at which the value is stored.
Examples:
Declare variable foobar with a value of 0 and printable with 2 digits after the decimal point: `[ "declare", "foobar", "0", 2 ]`
Increment foobar by 1: `[ "declare", "foobar", "foobar + 1" ]`
Use foobar in other expressions: `[ "if", "foobar > 42", [ "alert", "Foobar is as big as %s", "foobar" ] ]`

7.2.19 bottomframe

Sets the display mode of the frame at the bottom of the main window.
Takes one argument, 0 for ticker view, 1 for documentation view, 2 for gauge view.
Example: ["bottomframe", 0]

7.2.20 resetplot

When gauge panel has a live plot widget, this command clears all data from live plot and restarts plotting.
Takes no arguments.

7.2.21 haltplot

When gauge panel has a live plot widget, this command freezes the plot so no new data is drawn.
Takes no arguments.

7.2.22 procedure

Declares a procedure that may be called by name. The procedure can also call itself so this enables looping by recursion.
Takes two arguments, a name and the script to be run.
Example:
`[ "procedure", "fullthrottle", [ [ "confirm", "Are you sure?" ], [ "set", "throttletarget", 100 ] ] ]`

7.2.23 run

Calls procedure by name.
Takes one argument, the procedure name.
Example: `[ "run", "fullthrottle" ]`

7.2.24 endscript

Ends script execution and stops all asynchronous threads this script may have spawned.
Takes no arguments.

7.2.25 statusdialog

Asynchronously opens a dialog displaying status messages from script. Closing this dialog will end script execution by same means as endscript command.

Dialog will close by itself when script execution ends.
Optional arguments listed in order:

Initial status text String, defaults to empty.

Static text Boolean, defaults to false. If true, displays a text that doesn't change, more like a message box but asynchronous, script continues running while the dialog is open.

Font size Integer, sets the font size of the dialog. Default 16.

Title String, sets the title bar text of the dialog. Default "Script running"

Button label String, sets the label of the button that closes the dialog. Default is "Abort"

Examples:

```
[ "statusdialog" ]
[ "statusdialog", "This dialog will close when the
script stops", true ]
```

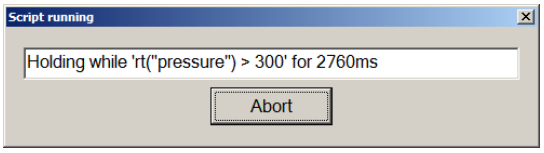


Figure 7.1: Script status dialog

7.2.26 status

Updates the status message displayed in the status dialog or other widget displaying the status of the current script.
Takes 0-9 arguments, if there's no argument the status is cleared, if there are arguments the first is a format string, same as confirm and other messaging functions that use format strings.
Example:

```
[ "status", "Waiting for engine to start" ]
```

7.2.27 onexit

Declares a callback that runs when the script exits either due to user interruption or other cause. Useful for resetting parameters that the script may alter during run time.
Takes one argument, the callback to run on exit.
Example:
```
[ "onexit", [ "set", "throttletarget", 0 ] ]
```

7.2.28 nop

No operation, no arguments.

7.2.29 edit

Opens a certain configuration item in the editor window.
Takes one argument, the id of the item to edit.
Example: `[ "edit", "mainfuelmap" ]`

7.2.30 rpc

Executes a remote procedure on the connected controller.
Takes one or more arguments, first being the identifier of the remote procedure. The rest being arguments if the remote procedure takes any.
If the remote procedure returns any data, those data values may be assigned to variables in the scripting memory space.
Returns false if no controller connected.
Examples:
Remote procedure that takes no arguments:
```
[ "rpc", "resetltft" ]
```
Remote procedure that takes input but returns no data:
```
[ "rpc", "canxmit", [ "widgetval(canid1)", "widgetval(plen1)",
"widgetval(b10)", "widgetval(b11)", "widgetval(b12)",
"widgetval(b13)", "widgetval(b14)", "widgetval(b15)",
"widgetval(b16)", "widgetval(b17)" ] ]
```

Remote procedure that returns data. Note that the return variables are placed within brackets at the end of the argument list:
`[ "rpc", "ccantest", [ "widgetval(ccoffset)", [ "cccanid", "ccplen", "ccb0", "ccb1", "ccb2", "ccb3", "ccb4", "ccb5", "ccb6", "ccb7" ] ]`

7.2.31 library

Executes a script from the script library provided by the configuration file. Scripts can be found under the "scripts" key in the definition section in the configuration file or they can be found under the "userscripts" key in the main part of the configuration file. If a script from userscripts bears the same name as a script from the definition section, the one from the userscripts section is used.
Example:
`[ "library", "canscan_interactive" ]`
If the description is omitted from the script definition, the script will not appear in the `Script library` section of the `Tools` menu of the application main window.

7.2.32 dialog

Opens a dialog, takes one or two arguments. Dialog layout is defined in the same manner as the gauge displays in the application, but with optional extra settings.

```
1  [ "dialog", {
2        "title": "Script dialog",
3        "xsize": 400,
4        "ysize": 300,
5        "noresize": 1,
6        "onclose": [ "alert", "Thank you for playing" ],
7        "bgcolor": "0xFFFFFF",
8        "fgcolor": "0x000000"
9     },
10    [
11    {
12       "id":"enginespeed",
13       "size":[
```

```
14          50,
15          20
16        ],
17        "position":[
18          50,
19          0
20        ],
21        "type":"gauge"
22      },
23      {
24        "caption":"Okay",
25        "size":[
26          50,
27          20
28        ],
29        "position":[
30          0,
31          0
32        ],
33        "type":"button",
34        "onclick": [ "rpc", "canxmit", [ 31337, 1, 42 ] ]
35      },
36      {
37        "caption":"This is a static text object",
38        "size":[
39          50,
40          20
41        ],
42        "position":[
43          0,
44          50
45        ],
46        "fgcolor": "0x0000FF",
47        "bgcolor": "0x0",
48        "type":"static"
49      }
50      ]
```

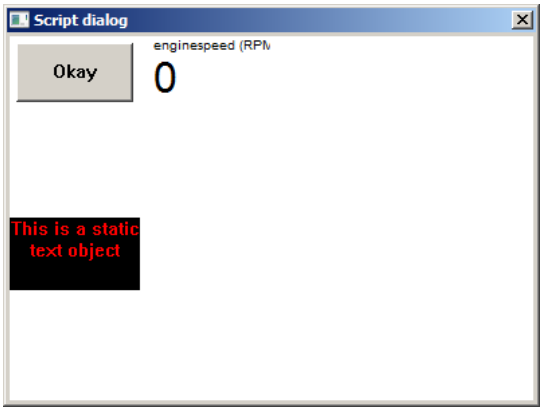


Figure 7.2: Script opened custom dialog

The optional dialog properties that may be specified:

xsize Specifies initial X dimensions. Default is 800 pixels.

ysize Specifies initial Y dimension. Default is 600 pixels.

bgcolor Specifies background colour in a hex string RGB format ("0xBBGGRR"). Default is black, "0x000000". This property is inherited to widgets unless they have their own bgcolor property specified.

fgcolor Specifies text (foreground) colour in the same manner as the bgcolor property and is inherited in the same manner. Default is white, "0xFFFFFF".

noresize Boolean option, if set to 1 or true will prevent the window from being resized or maximised.

title Sets the window title. Default is "Gauge dialog".

maximised Specifies that the dialog should start maximised. xsize and ysize properties still apply if the user unmaximises the dialog.

onclose Specifies a script to be run when the dialog is closed.

7.2.33 burn

Saves configuration to controller flash memory. Returns false if no controller connected or operation fails for any reason.

7.2.34 getcheck

When using a dialog box with a button widget that is either stateful (sticky) or checkbox type, this instruction gets the state of the button. Takes one to three parameters, first being the widget identifier. Second parameter specifies a script to run if button is in pressed or checked state, third parameter specifies a script to run if button is not in pressed or checked state. If only the widget identifier is specified, the function returns 1 if button is in pressed/checked state and 0 if not. If scripts are specified then the return value of the script is passed back. Returns 1 if button unpressed and a second parameter is specified but no third parameter.
Example:

```
[ "getcheck", "foobar", [ "alert", "Button is pressed"
], [ "alert", "Not pressed" ] ]
```

7.2.35 setcheck

When using a dialog box with a button widget that is either stateful (sticky) or checkbox type, this instruction sets the state of the button. Takes two parameters, first being the widget identifier and the second being the button state. A state value of 0 specifies that the button is up and not checked, a state of 1 specifies that the button is down or checked. Button events (`oncheck, onuncheck, onclick`) are not generated when button state is altered by the script.
Example:

```
[ "setcheck", "foobar", 1 ]
```

7.2.36 settext

When using an edit widget or a static text widget, this instruction sets the text displayed by the widget. It takes two or more arguments, first one being the widget identifier, the second being a format string and the rest according to the same convention as other string formatting instructions.
Example:
`[ "settext", "foobar", "%s", "enginespeed" ]`

7.2.37 isediting

Conditional test that takes one argument which is the identifier of a configuration item and returns true if application is currently in edit mode and the item being edited matches the provided identifier. Also takes optional second and third arguments which are scripts for further execution depending on whether the condition is true or false.

7.2.38 refresheditor

Refreshes data shown in editor, if main dialogue is currently in edit mode. Used if altering configuration programmatically to make sure the editor is displaying a current version. Takes no arguments. Always returns true.
Example:
`[ "isediting", "mainfuelmap", [ "refresheditor" ] ]`

7.3 Script examples

7.3.1 Dyno sweep test

```
1 [
2   [ "statusdialog" ],
3   [ "status", "Waiting to settle" ],
4   [ "set", "speedtarget", "c(\"initialspeed\")" ],
5   [ "onexit", [ "set", "throttletarget", 0 ] ],
6   [ "set", "throttletarget", 100 ],
7   [ "while", "rt(\"enginespeed\") < c(\"speedtarget\")", [ "nop" ],
```

```
8       [ "hold", "abs(rt(\"enginespeed\") - c(\"speedtarget\")) < 100", 2000,
9         [
10          [ "status", "Running sweep" ],
11          [ "set", "accelmode", 1 ], // Start acceleration
12          [ "while", "rt(\"enginespeed\") < c(\"endspeed\")", [ "nop"] ]
13        ]
14      ]
15    ]
16  ]
```

7.3.2 Slowly increment throttle

This program increments config value throttletarget at the rate of 50% per second if the application is running at 50 frames per second.

```
1  [
2    [ "statusdialog" ],
3    [ "while", "c(\"throttletarget\") < 100",
4      [
5        [ "set", "throttletarget", "c(\"throttletarget\") + 1" ],
6        [ "status", "Throttle at \%s percent", "throttletarget" ]
7      ]
8    ]
9  ]
```