EFI Analytics ECU Definition files

Authors

Phil Tobin

Revisions

Date	Change	By	Version
September 14, 2016	Initial Release	Phil Tobin	0.8
September 15, 2016	Added sections on [ReferenceTables] and [LoggerDefinitions] Fixed errors.	Phil Tobin	0.9
September 21, 2016	Fixed missing parameterStartOffset definition.	Phil Tobin	0.91
September 22, 2016	Updated for nested #define and added getByOffset function definitions.	Phil Tobin	0.92
September 22, 2016	Added information on password protected dialogs	Phil Tobin	0.92.1
September 28, 2016	Updated spelling / grammar errors	Phil Tobin	0.93
October 4, 2016	Added section on High Speed Run-time	Phil Tobin	0.94
October 7, 2016	Added 3.49 support for xyLabel on TableEditor	Phil Tobin	0.95
November 19, 2016	Added how to define dynamically resized TableEditor views sec 4.4.5.5.1	Phil Tobin	0.96
January 12, 2017	Added indicatorPanel usage to [UiDialogs] section	Phil Tobin	0.97
June 19, 2017	Added additional information on using cached PDF's.	Phil Tobin	0.98
October 25, 2017	Updated with info on resizable tables.	Phil Tobin	0.99
January 12, 2018	Updated to include ini extensions through iniSpec 3.58. New Functions and verticalMarker for High Speed Loggers. Added FTPBrowser usage	Phil Tobin	1.0
March 11, 2018	Added support for portEnabledCondition in PortEditor definition.	Phil Tobin	1.01
April 2, 2018	Added support for DataLogView ini section	Phil Tobin	1.02
Jan 15, 2019	Update new PcVariable extension uses.	Phil Tobin	1.03
Jan 28, 2019	Added instructions for making ControllerCommands visible to users.	Phil Tobin	1.04

June 28, 2019	Updated added iniSpec features for 3.64	Phil Tobin	1.05
July 8, 2019	Updated iniSpec for 3.65	Phil Tobin	1.06
Sept 26, 2019	Updated iniSpec for 3.66	Phil Tobin	1.07
March 6, 2020	Updated arrayValue function to specify use of the array. Prefix on constants	Phil Tobin	1.08
Nov 3, 2021	Updated to include new ini attributes up to iniSpec 3.69. Added information on using UDP Streams for high speed loggers to the LoggerDefinition section.	Phil Tobin	1.09
Nov 3, 2021	Included usage of short hand bit field option definition for ECU Parameters	Phil Tobin	1.10
Nov 5, 2021	Added additional info on how dynamically sized arrays are managed in section 4.4.5.5.1 Added sample rendering of UDP Stream data	Phil Tobin	1.11
May 16, 2022	Documented support for readoutPanel, readout and runtimeValue UI entries.	Phil Tobin	1.12
Dec 19, 2022	Updated to include 3.73 ini updates.	Phil Tobin	1.13

Contents

1 Summary 6
2 [TunerStudio] 8
2.1 Attributes 8
3 Command Set 13
3.1 General 13
3.2 Standard Commands 13
3.3 Read / Write Operations 13
3.4 Keywords and Setup 16
4 [Constants] 19
4.1 Summary 19
4.2 Expressions 19
4.3 Page 20
4.4 Attributes 20
4.5 Last Attributes Keywords 25
5 [PcVariables] 27
5.1 Specialized PcVariables: 27
6 [ConstantExtensions] 29
6.1 Keywords 29
7 [SettingGroups] and Directives 32
7.1 Overview 32
7.2 Usage 32
7.3 Additional Directives 33
8 [ControllerCommands] 35
8.1 General 35
8.2 User Displayed Controller Commands 35
9 [CurveEditor] 36
9.1 1D Array Graph Editors 36
9.2 Entry Syntax 36
9.3 Example Curve Editors 37

10 [TableEditor] 42
10.1 Overview 42
10.2 Syntax: 43
10.3 Resizable Tables 44
11 [OutputChannels] 46
11.1 Overview 46
11.2 Optimized OutputChannel capture 47
12 [GaugeConfigurations] 50
12.1 Gauge Categories 50
12.2 Gauge Templates 50
13 [Datalog] 52
14 [Menu] 53
14.1 Standard Dialogs 53
14.2 Defining Menus 54
14.3 Example Menu Definition 56
15 [UserDefined] – [UiDialogs] 57
15.1 Overview 57
15.2 Section keywords 57
15.3 User Help 58
15.4 Defining Dialogs 59
15.5 Dialog Examples 79
16 [FrontPage] 82
16.1 Example FrontPage 83
17 [KeyActions] 83
18 [LoggerDefinition] 85
18.1 Overview 85
18.2 Example LoggerDefintions 88
19 [PortEditor] 92
20 [ReferenceTables] 95
20.1 Overview 95
20.2 Commands 95

20.3 Defining a Reference Table 95
20.4 Example entries: 97
21 [SettingContextHelp] 100
22 Expressions and Math Functions 101
22.1 Expressions 101
22.2 Operators 101
22.3 Functions 102
23 String Functions 105
23.1 Overview 105
24 [FTPBrowser] 106
24.1 Defining an FTP Browser: 106
25 [DatalogViews] 108

1 SUMMARY

EFI Analytics ECU Definition files, often known as .ecu or .ini files contain all information needed to interact with a controller and present the information to users with defined views for a simpler, non-Engineer usage. This document reviews each non-proprietary section to describe it's function and how it is used. For proprietary components, these components will be documented elsewhere.
Key file components:

- Controller Memory mapping
- Protocol commands (Basic Request Reply Protocol)
- Runtime data definitions
- Menu Options
- UI Dialog Definitions
- Data Log fields
- Gauge Template definitions

This provides the controller firmware developer full control of not just data exchange, but how settings will be presented and modified while maintaining the feel of a fully integrated application.

Each section is loaded in line item order.

The load order for these sections is:

1. [TunerStudio]
2. Load all non-section specific commands and entries such as protocol commands and memory space
3. [PcVariables]
4. [Constants]
5. [OutputChannels]
6. [Replay]
7. [ExtendedReplay]
8. [TableEditor]
9. [GaugeConfigurations]
10. [ControllerCommands]
11. [PortEditor]
12. [CurveEditor]
13. [TriggerWheel]
14. [UiDialogs] / [UserDefined]
15. [FTPBrowser]
16. [Menu]
17. [KeyActions]
18. [Datalog]
19. [FrontPage]
20. [VerbiageOverrides]

21. [ConstantExtensions]
22. [TurboBaud]
23. [EventTriggers]
24. [VeAnalyze]
25. [WueAnalyze]
26. [Tools]
27. [LoggerDefinition]
28. [SettingContextHelp]
29. [DatalogViews]

2 [TunerStudio]

2.1 Attributes[1]

2.1.1 signature

A signature is required in every ECU Definition file. It is used to identify and match the ECU Definition to the firmware reported signature.

Example

```
signature     =    "Firmware Prefix 1.51.02 ";
```

During initial interrogation of the controller, the signature will be requested.

If using an XCP transport, the GET_ID command will be issued in mode 1

For basic request reply protocol, the defined queryCommand command will be used.

```
queryCommand   = "r\$tsCanId\x0f\x00\x00\x00\x14"  ; Verify against signature.
versionInfo    = "r\$tsCanId\x0e\x00\x00\x00\x3c"  ; Title bar, this is the code version.
```

The resulting signature string will be used to match the current firmware with the ECU Definition file required to interact with the controller.

2.1.2 iniVersion

iniVersion is to track changes and updates to the ECU Definition file. This allows a file to be identified as an updated ECU Definition file while still for the same firmware. This is an optional attribute. A file with any definition number assigned will be assumed to be more current than one without.

Applications may differ on how they treat this attribute.

There are 2 ways this can be treated depending on application configuration:

- Look for a newer iniVersion file when creating a project.
- Automatically update the ini in your project if the iniVersion of the installer for the same firmware / signature is lower or not defined.

2.1.3 helpManualDownloadRoot

Defines a root URL that help documents for the firmware related to the ECU Definition file are located. This can be referenced in help links added to Calibration dialog help menus.

helpManualDownloadRoot = http://www.somedomain.com/doc/pdf/

The Application will cache pdf help documents to the local computer for offline access, however, if the document is not yet available on the local computer, it will check for the pdf at this url. See User Help .

2.1.4 iniSpecVersion

Within the [TunerStudio] section the iniSpecVersion is defined:

```
iniSpecVersion = 3.46
```

This number is increased when new ini features become supported and it is the applications responsibility to check this value upon ini load to insure it does support all features used within the ini file. EFI Analytics Calibration tools will report an error that the user must upgrade the application if the iniSpecVersion found in the ini file is greater than what that version of the application supports.

```
Recent ini feature to version
2.9 added support for:
- F32 in constants and OutputChannels
- Turbo Baud to activate during SD download
- Quick runtime read data "g" command.
3.0 added Support for:
- Hide menu expressions
- GetChannelValueByOffset, getChannelScaleByOffset, getChannelTranslateByOffset
- ConstantsExtensions - useScaleAsDivisor = Constant, { expression }
- Constant Digits expression
- channelSelector widget
3.1
- PortEditor support of activateOption = extendedDataInSize
- PortEditor support of activateOption = filter32BitChannels
- Support for fullTimeTurboEnabled, turn up baud as soon as connected.
3.11
- Added support for canDeviceSelector
3.12
- Added support for nextOffset keyword instead of hard number on Constants and OutputChannels.
3.13
- Added support for lastOffset keyword instead of hard number on Constants and OutputChannels.
3.2
- supports visibility on panels.
- basic expression based table sizing
3.21
- Support for Curve Lines to be added via expressions
- Support in string functions i.e "Cyl $stringValue( Fire_Ordr_High[0])"
- Support for string function in Help Topics.
- Support for string function in Row Item Label; fields.
3.22
- Support for maintainConstantValue in [ConstantsExtensions]
    ;maintainConstantValue = [someConstant],    { some expression resolving to desired constant value }
    maintainConstantValue = stoich,    { selectedFuel ==1 ? 9.86 : 14.7 }
3.23
- Support for String expression in help lines
- New String Function $getProjectsDirPath()
- New String function $getWorkingDirPath()
3.24
- Added helpManualDownloadRoot = "http://www.efianalytics.com/TunerStudio/docs/"
```


3.25
- Added parameterStartOffset pageable
- Added outputChannelStartOffset
3.26
- Added support for Replay section
3.27
- Added Support for oppositeEndian
3.28
- Added Support for controllerPriority - Constant Flag, Will save and load from msq, but always silently update from the controller on connect.
- Added ExtendedReplay section
- OutputChannel hidden now based on expression in addition to hidden flag.
3.29
- Added [EventTriggers] section
- Added support for timedPageRefresh = [pageToRefresh], [timePeriodBetweenRefreshesInMs Expression]
3.30
- Added Support for showTextValues on CurveGraphs
- Added Visible Condition to Main Menu's
3.31
- Added Support for showPanel Key Actions.
3.32
- Added Support radio on bit fields.
  ;radio = [horizontal|vertical], "Label Text" , [ConstantName]
  radio = horizontal, "Master Enable" , can_poll
- Support for controllerPriority in ConstantsExtension section.

3.33
- Added support for EcuUiField to display values in Hex
  ;field = "Label Text", [Constant], {enabledExpression}, {visibleExpression}, displayInHex
  field = "Internal Value", int_val, {1}, {1}, displayInHex
- Added support for horizontal and vertical radio
3.34
- Added useMappingTable = constant, mappingFile.inc; Allows for scaling based on a lookup inc both ways.
  Entry in [ConstantsExtensions] section, set scale in Constants Section to 1.0;
  ;useMappingTable = afrBins1, WBafr100Zeit.inc
3.35
- Added std_bootstrap Standard Dialog for BS3
3.36 - 2015-05-23
- Added Support for triggeredPageRefresh, Ex:
 triggeredPageRefresh = 4, { AFRStoich0 != afrStoich0 || AFRStoich1 != afrStoich1}
3.37 - 2015-06-10
- Fix number of parameters for single Array tableLookup function.
3.38 - 2015-06-15
- Added support for indexCard layout.
3.39 - 2015-07-08
- Added support for closeDialogOnClick to commandButtons
3.40 - 2015-10-20
- Added support for expressions in filter qualifying value
- Added support for string expressions on indicator text

3.41 - 2015-12-16
- Added Support for rawValue as a default value in ConstantExtensions
3.42 - 2016-03-05 Added Support for showXYDataPlot on CurveGraphs
3.43 - 2016-03-17 Added Support for M08 data type, behaves like unsigned byte until maximum, then becomes a negative signed byte.
3.44 - 2016-03-22 Added Support for delayAfterPortOpen - this will add a delay after opening port before sending any communications.
3.45 - 2016-04-18 Added Support for canClientIdSelector.
canClientIdSelector = "Remote CAN ID", can_poll_id2, {can_poll && (enable_pollPWM)}
3.46 - 2016-07-20 Added support for OutputChannel or expressions for maximum bytes in logFieldSelector
3.47 - 2016-09-01 Support for #define String Lists used as bit options with over-rides.
3.48 - 2016-09-22 Support for nested #define String Lists within #define String Lists.
3.49 - 2016-10-07 Support for xyLabels on [TableEditor] definitions.
3.50 - 2016-10-26 Support for hex integer values on many parameters (pageSize, parameterStartOffset, etc).
3.51 - 2017-01-11 Support for indicatorPanel added to UserDefined
3.52 - 2017-05-15 Support nextOffset and lastOffset for internalLogFields.
3.53 - 2017-10-26 Support for maximumElements added. This will limit the amount of memory to be used by a resizable table.
3.54 - 2017-12-08 Added Vertical Marker for LoggerDef
Added translate to LoggerRecord
3.55 - 2017-12-13 Added support for [FTPBrowser] ini section
3.56 - 2018-01-06 Added support for arrayValue function
3.57 - 2018-01-08 Added support for selectExpression function
3.58 - 2018-01-15 Support for PortEditor outputName.
3.59 - 2018-03-11 Added support for PortEditor portEnabledCondition
3.60 - 2018-04-02 Added support for DatalogViews Ini Section
3.61 - 2018-05-15 Added support for TuningViews & EncodedData Ini Section
3.62 - 2018-06-07 Added new PcVariable paramClasses to follow and store an OutputChannel
3.63 - 2019-01-10 Support String functions dynamically changing table and dialog title
3.64 - 2019-06-28 Added support for suppressing the graph on CurveEditors. Allows for chart only display
3.65 - 2019-07-08 Added support for forceBigEndianProtocol. When set to true, big endian will be used for protocol offset and len regardless of type endian set for data.
3.66 – 2019-09-26 Added support for overlaidDataSetCount in the LoggerDefinition section.
3.67 - 2019-10-23 Added support for defaultXAxis in LoggerDefinitions, for generic Data this will be the initial X Axis.
3.68 - 2020-10-27 Added support for readSdCompressed = [true/false] - Default is true.
3.69 - 2021-10-19 Added support for noCommReadDelay = [true/false] -
Default is false; true disables legacy delays optimized for 115200 baud devices.
Added support for defaultRuntimeRecordPerSec = 15; Changes the default data rate.
Valid integer values are those in the Data Rate dialog.
Added support for defaultIpAddress = 192.168.1.80 ; your desired IP4
Added support for defaultIpPort = 2000; any valid integer
Added support for bit options defined for specific values
All undefined indexes will be filled with "INVALID". Example:
1="Option 1", 3="Option 2"
equivalent of: "INVALID", "Option 1", "INVALID", "Option 2"
3.70 – 2021-11-15 Added an optional attribute in the Constant definition allowing you to over-ride the default column width.
3.71 - 2022-01-18 - Added support for portActiveDelay and portInactiveDelay in the port editor section.

These fields are optional 1D arrays that will be set to the delay in changing Pin state after the condition becomes true.

3.72 - 2022-05-16 Added support for readoutPanel and readout added to UserDefined.
Added support for runtimeValue as a field. Same as field Definition, but give it an OutputChannel name instead of Constant.
Added support for ignoreMissingBitOptions = true, by default false. When true, warnings of missing bit options will be suppressed, missing elements will be filled with "INVALID"

3.73 – 2022-12-19 Added support for stopOnExit in the LoggerDefinition section. when set to true, the logger will be stopped when leaving the view. In tunerStudio, this would be when the High Speed Logger tab is left.
Added support for DataLogField Categories.

3 COMMAND SET

3.1 GENERAL

If using a Basic request reply protocol, the commands used for common tasks are defined. However, there are several key definitions required for proper memory mapping. This is generally in the [Constants] section, but can be anywhere in the ecu/ini file if that keeps better order based on the command function.

3.1.1 Command Formatting

commands are comprised of String represented bytes delimited by a backslash \
In the case of the versionInfo Command: "r\$tsCanId\x0e\x00\x00\x00\x3c"
r – an ASCII r will be used 0x72
$tsCanId – this will be substituted with the Constant value tsCanId
x0f – hex value 0xF
…
The final bytes to be sent for this command assuming the Constant value of tsCanId = 0:
byte[0] = 0x72
byte[1] = 0x00
byte[2] = 0x0E
byte[3] = 0x00
byte[4] = 0x00
byte[5] = 0x00
byte[6] = 0x3C

3.2 STANDARD COMMANDS

3.2.1 Standard Identification Commands

```
versionInfo    = "Your Command Here"  ; Title bar

queryCommand   = " Your Command Here "  ; Verify against signature.
```

3.2.1.1 MegaSquirt 3 Example

```
versionInfo    = "r\$tsCanId\x0e\x00\x00\x00\x3c"  ; Title bar
queryCommand   = "r\$tsCanId\x0f\x00\x00\x00\x14"  ; Verify against signature.
```

3.3 READ / WRITE OPERATIONS

3.3.1 nPages

Required for XCP or request reply protocol

Defines the number of pages in the memory mapping.

Example, Define 3 pages:

```
nPages          = 3
```

3.3.2 pageSize

Required for XCP or request reply protocol

A comma delimited set of integers defining the page sizes in order. The number of Integers provided must match nPages set. Pages do not have to represent actual pages in the controller. Pages can be used to sub divide the calibration data to work with smaller chunks or to reorder defined Constants to respect dependencies.

Example – 3 pages all 1024 bytes long:

```
pageSize        = 1024,         1024,         1024
```

3.3.3 parameterStartOffset

Optional = Yes.

If defined, a memory start address is defined for each page. This start offset will be added to the Constant offset for the final address of any given constant. If not defined the parameterStartOffset is assumed to be 0.

Address = parameterStartOffset[page] + constantOffset

3.3.4 pageIdentifier

pageIdentifier is not required. This is purely optional and to define a standard reference or prefix to a command. In other read, write and burn commands this prefix can be referenced with a %2i notations.

It is always expected to be 2 bytes, thus the %2 notation with I representing identifier.

3.3.5 burnCommand

The Burn Command is the command to be sent to the controller to initiate a write to flash. There should be a burnCommand defined for each page. If the firmware does not support a burn command for a specific page, the command for that page should be an empty string.

By default, a burn command will be sent when any of the following occur:

- A settings dialog is closed.
- A “Burn” button is clicked.
- There is data written to a page other than the last page written to.

Automatic burn can be disabled at the applications discretion by editing the application properties file by adding the following entries:

```
# Change to false if you want either auto burn disabled. This is only recommended for advanced
# purposes such as firmware development.
# autoBurnOnCloseDialog - if true(default) a burn command will be sent when a dialog is closed
#     insuring all changes have been persisted to the controller.
# autoBurnOnPageChange - if true(default) a burn command will be sent for the last write page
#     when a command to write data to a new page is received. This prevents writing to a new
#     page until all writes have been persisted
autoBurnOnCloseDialog=true
```

autoBurnOnPageChange=true

3.3.6 pageReadCommand

Optional – Required for all legacy protocols. An “XCP” per page notation required for XCP.

This command will be sent to request a read of a full data page. During interrogation full pages will be read, thus a request of this type will be sent for each defined data page. It is expected that there will be a command defined for each page that was set by nPages.

3.3.6.1 Request Reply Protocol Example:

pageReadCommand = "r%2i%2o%2c", "r%2i%2o%2c", "r%2i%2o%2c"

Comma delimited command per page.

Command breakdown:

r	ASCII ‘r’ or hexadecimal 0x72, any ASCII or hex value is valid.
%2i	This will be replaced with 2 bytes defined in pageIdentifier. This is optional, and can be any number of hard bytes defined with standard notation \xXX\xXX\xXX
%2o	This notation will be replaced with a 2 byte offset/memory address of the start address. This will be parameterStartOffset+offset. The value of parameterStartOffset is 0 by default but can be defined using parameterStartOffset
%2c	Will be replaced with a 2 byte representation of the length in bytes to be read.

Expected Response:

The bytes from page %2i, starting at offset %2o and of length %2c.

3.3.7 pageValueWrite

A defined Command to write a single byte.

Optional – Yes. If not defined, will always use pageChunkWrite. However, either pageChunkWrite or pageValueWrite Must be defined for calibration. If there is no pageWriteChunk, bytes will be written 1 at a time.

Example:

pageValueWrite = "w%2i%2o%2c%v", "w%2i%2o%2c%v", "w%2i%2o%2c%v

3.3.8 pageChunkWrite

A defined command to write a block of bytes to a Controller.

w	ASCII ‘w’ or hexadecimal 0x77, any ASCII or hex value is valid.

%2i	This will be replaced with 2 bytes defined in pageIdentifier. This is optional, and can be any number of hard bytes defined with standard notation \xXX\xXX\xXX
%2o	This notation will be replaced with a 2 byte offset/memory address of the start address. This will be parameterStartOffset+offset. The value of parameterStartOffset is 0 by default but can be defined using parameterStartOffset
%2c	Will be replaced with a 2 byte representation of the length in bytes to be written.
%2v	Will be replaces with a byte array of %2c in length containing the values to be written.

3.3.9 crc32CheckCommand

Optional – Yes. When called, it is expected that a standard CRC32 is returned for the entire page.

If this is defined, it can be used to verify that the application local store matches what is on the controller. Availability of this command can significantly speed interrogation time to the user.

The format of this command is entirely up to the firmware implementation and can be defined using the standard notations within this document.

3.3.10 outputChannelStartOffset

Optional.

Defines a base address to be added to the %2o notation when reading runtime data using non-XCP block reads. By default this value is 0, thus the offset will be the index in the stream, when set to a value

%2o = outputChannelStartOffset + streamOffset

Stream offset for each runtime value is defined in the [OutputChannel] section.

3.4 KEYWORDS AND SETUP

3.4.1 Endianness

endianness - define byte order of the MCU for multi-byte data types. This defaults to Motorola style Big Endianness, but can be used to set little. Valid values are big and little. This is not used with XCP, the XCP standard is followed.

Options:

endianness = big

endianness = little

3.4.2 blockingFactor

Optional, does not apply to XCP where Max CTO and Max DTO will be honored

Limits the largest chunk to be written or read. This limit applies to calibration data and runtime data reads. This limit includes the payload data only, it does not count the commands toward the count.

blockingFactor = 256

Will limit read and write blocks to 256.

3.4.3 interWriteDelay

Time in ms to wait between each byte written to the controller.

In cases where there is no DMA or the MCU can only read incoming data so quickly without risk of overflowing or overwriting the buffer, this will throttle the speed that it is written.

Example:

interWriteDelay = 3

This will create a 3 ms wait between each byte written to the controller. Data will still be read from the controller as fast as possible.

3.4.4 tsWriteBlocks

Applications will write to the buffer at full speed with no wait between bytes. interWriteDelay will be ignored.

Default: off

tsWriteBlocks = on ;

3.4.5 blockReadTimeout

For Basic Protocols, this defines the maximum time in ms the application will wait for response data from the controller before timeout.

Example:

blockReadTimeout = 250

3.4.6 pageActivationDelay

Sets a wait time in ms after activating a data page before any read / write operations will be performed.

3.4.7 messageEnvelopeFormat

Defines any proprietary envelopes to wrap each command in. The envelope is implemented by the application, but activated in the ini.
Applies to XCP: No

Optional: Not required if there is no envelope
Possible values:
messageEnvelopeFormat = msEnvelope_1.0

To read more on the MegaSquirt envelope used by MS3 1.1 and up and MS2 Extra 3.3+ see:
http://www.msextra.com/doc/pdf/Megasquirt_Serial_Protocol-2014-10-28.pdf

3.4.8 refreshLocalStoreOnActivity

By default the application will read cal data from the controller whenever a user calls certain views into focus such as opening a settings dialog. This gives fall back insurance that the data on the controller is always what is displayed even in the event there was a reset or failed write. This can be disabled using:

```
refreshLocalStoreOnActivity = false
```

4 [CONSTANTS]

4.1 SUMMARY

The primary goal of this section is to map the controller calibration data to referenced variables with Meta data attributes:

- Naming
- Memory location – Numeric or keyword
- Parameter class - Keyword
- Data Type – Keyword
- Shape / bits
- Units – String or StringFunction
- Scale – Numeric or Expression
- Translate – Numeric or Expression
- Limits – Numeric or Expression
- Display precision – Numeric or Expression

The application will enforce and set limits and update presented values automatically when any dependent components are updated.

Formatting of a Constant entry:
name = class, type, offset, shape, units, scale, translate, lo, hi, digits

4.2 EXPRESSIONS

In the definition of Constants, any numeric component can be replaced with an expression.

For example, this can allow you to display a temperature in °F or °C based on a set user preference.

Expressions are built using the values of other Constants, PcVariables or OutputChannels combined with supported functions and operators to produce values for each attribute. This allows the values to be dynamically set based on other conditions and preferences. Also See Expressions and Math Functions

Expressions can be used in in most cases instead of hard numeric values. This is common with scale and translate

Constant values are loaded from calibration files in the order they are defined in the ECU definition. Therefore, any referenced Constant must be defined prior to prevent changes in values.

Expressions in Constant definitions are evaluated dynamically as user settings are entered at runtime, instead of at ini load time and take effect as the user is configuring the controller. The variable {afrMax} (See Below) can be a Constant, PcVariable or any expression build from them. This same syntax can be applied to scale, translate, digits, minimum or maximum value. The Standard comparators can be used within an expression.

You can also use compound conditions separated by:

&&

||

This is a simple use of an expression; however, an expression can be as complex or as simple as you need. In this case we are using a PcVariable named “afrMax” within the expression brackets to set the max value on the constant “afrTable1”. The PcVariable “afrMax” can now be used in a UI dialog so that when a value is entered it will set the max value allowed in the Z axis cells of afrTable1.

```
 ; name     = class,  type, offset,  shape,  units,  scale,    translate,    min,    max,    digits
afrTable1  = array ,  U08,   48,  [12x12],  "AFR",    1.00,    0.00000,    9.00,   {afrMax},    1
```

In-line evaluation

In some instances, you may want different calculations performed based on a condition. This can be performed with an in-line evaluation. In-line evaluations are commonly used in the [OutputChannels] section of the ini file to evaluate a min or max parameter for a constant.

```
  afrMax    =     {  condition  ?  true expression  :  false expression}
    1.                   2.            3.                   4.
```

The constant or PcVariable in this location will be set equal to the evaluated expression

Here you will use a bit constant to switch between the two optional expressions

bit 0 = true any other bit value used will equal false.

1 – The new Channel name

2 – the Condition expression. May be a single constant or complex expression.

3 – This expression will be used if the condition of the condition expression at position 2 is true.

4 – This expression will be used if the condition of the condition expression at position 2 is false.

4.3 PAGE

At least 1 page is required. The start of a page is denoted by a row with a page tag followed by the page number.

page = 1

Page numbering starts with 1 in the ECU Definition.
Note: with XCP page numbering will start with 0, so in the ECU Definition the page will be assigned a number 1 greater.

4.4 ATTRIBUTES

4.4.1 Name

The Constant name can be made up of alpha numeric characters and must start with a letter. Additional characters such as _ are allowed, no mathematical operators should be used.

4.4.2 Parameter Class

There are 5 parameter class types. The set type will establish the basic characteristics of the defined Constant and required attributes.

Parameter Class	Required attributes
bits	type, offset, list of indexed options
scalar	type, offset, units, scale, translate, min, max, digits
array	type, offset, units, shape, scale, translate, min, max, digits
String	type, offset, length, displayColumns
oddArray	type, offset, units, shape, scale, translate, min, max, digits

4.4.3 Type

Type defines the byte size of the Constant. Valid values include:

- U08 – Unsigned byte
- S08 – Signed Byte
- U16 – Unsigned Word
- S16 – Signed Word
- U32 – Unsigned DWORD
- S32 – Signed DWORD
- F32 – 32 bit floating point
- ASCII – For String param class

4.4.4 offset

The offset is an integer representing the distance from the start of the page to the 1st byte in the constant.

Keywords nextOffset and lastOffset are also valid.

nextOffset – Automatically moves to the next position after the last defined Constant. Thus if the last defined constant had an offset of 21 and is a 2 byte type U16, nextOffset will now resolve to 23.

lastOffset – Will resolve to the same offsetValue of the last defined Constant. This is commonly used when defining bit fields or multi-mapping memory.

At the beginning of each page, lastOffset and nextOffset will both be set to 0.

4.4.5 Shape

Shape is for parameter class bits and array. The notation and purpose is different in each instance.

4.4.5.1 bits

Shape defines the bits of interest within the byte using the notation: [lowBit : highBit] where lowBit represents the least significant bit being used, highBit represents the most significant bit.

Example:

[0:3] – Represents using the lower nibble of the byte with 16 possible options, the upper nibble is still available for other purposes – XXXX1111

[3:3] – Only bit 3 is of interest – XXXX1XXX

4.4.5.2 array

Shape defines the 1d or 2d array shape or the number of rows and columns in the table.

[8] – A 1D Array with 8 elements will be created.

[12x12] – A 2D array with 12 columns and 12 rows for a total of 144 elements will be created. Dimensions are defined as [columns x rows]

4.4.5.3 bits

Are numbered 0 – 7, the rightmost being zero. The basic data word that stores bit fields must be unsigned. You do not need to have a matching number of labels for the number of bits that you have specified. If a greater number of bits is allocated than the number of labels needed the remaining unused bits will need to be set to "INVALID" or the remaining bit values will be displayed in the drop down box as a numeric value. If no text strings are provided, then the drop down box will display the bit values starting at 0 and count up consecutively to the largest value available in conjunction with the number of bits allocated.
In some applications you may want the bit values to start at 1 instead of 0; this can be achieved with this notation [0:2+1], this will display 1 through 4 in the drop down box. Below is a few examples.

1 bit needs 2 values	5 bits need 32 values
2 bits need 4 values	6 bits need 64 values
3 bits need 8 values	7 bits need 128 values
4 bits need 16 values	8 bits need 256 values

The bit param class is used to set a discrete value within part of a byte and present these options to a user as a drop down list or radio button of String options.

```
; name     = class, type, offset,   shape,  [Option1], [Option2], etc.
myConstant   = bits,    U08,      0,      [0:2], "INVALID","One","Two","Three"
```

The Application will suppress any "INVALID" entries from being presented or selected.

If there are less entry options than possible options, the application will fill with the numeric values.

For long redundant option lists see #define

An alternate, short hand way to define the list of valid bit Options without needing to place the redundant "INVALID" options:

```
myConstant   = bits,    U08,      0, [0:2],  1="Option 1", 3="Option 2"
```

This is the equivalent of:

```
myConstant   = bits,    U08,      0, [0:2],  "INVALID", "Option 1", "INVALID", "Option 2"
```

4.4.5.4 scalar

A single numeric value that can be scaled and translated between the raw byte value and user/human value.

```
myScalar  = scalar,  U08,    1,   "",  0.019608,  0.00000,  1.00,  5.00,    2 ;
```

myScalar will be a single byte variable with a range of 0-5 and will be displayed with 2 decimal places.

4.4.5.5 array

Arrays are specified just like scalars, except that they have a "shape" entry in the fourth parameter. The shape allows you to define lists or tables, for example [8] defines a list with eight values and [2x4] defines a table with eight values (two rows and four columns). Tables may be stored in either "X-" or "Y-order." X-order means that memory is mapped.

[x1,y1] [x2,y1]...[xn,y1] [x1,y2]...

Y-order would be

[x1,y1] [x1,y2]...[x1,yn] [x2,y1]...

To use the TableEditor, you must define two lists and a table, and the lengths of the lists must correspond to the shape of the table.

Arrays can be 1D or 2D and will take byteSize * dimension1 * dimension2 bytes in length. Thus a U16 Type 2D array of 12x12 will require 288 bytes.

Indexing is row by row.

mySingleArray = array , U08, 346, [8], "ms", 0.10000, 0.00000, 0.00, 25.50, 1

This entry would create a single array 8 elements long of unsigned bytes with a range of 0-25.5 and 1 decimal place will be displayed. Units are set to ms

myDoubleArray = array , U08, 192, [12x12], ":1", 0.10000, 0.00000, 1.00, 25.00, 1

This entry would create a double array 12x12 elements long of unsigned bytes with a range of 0-25.0 and 1 decimal place will be displayed. Units are set to ":1".

4.4.5.5.1 Dynamically sized Arrays.

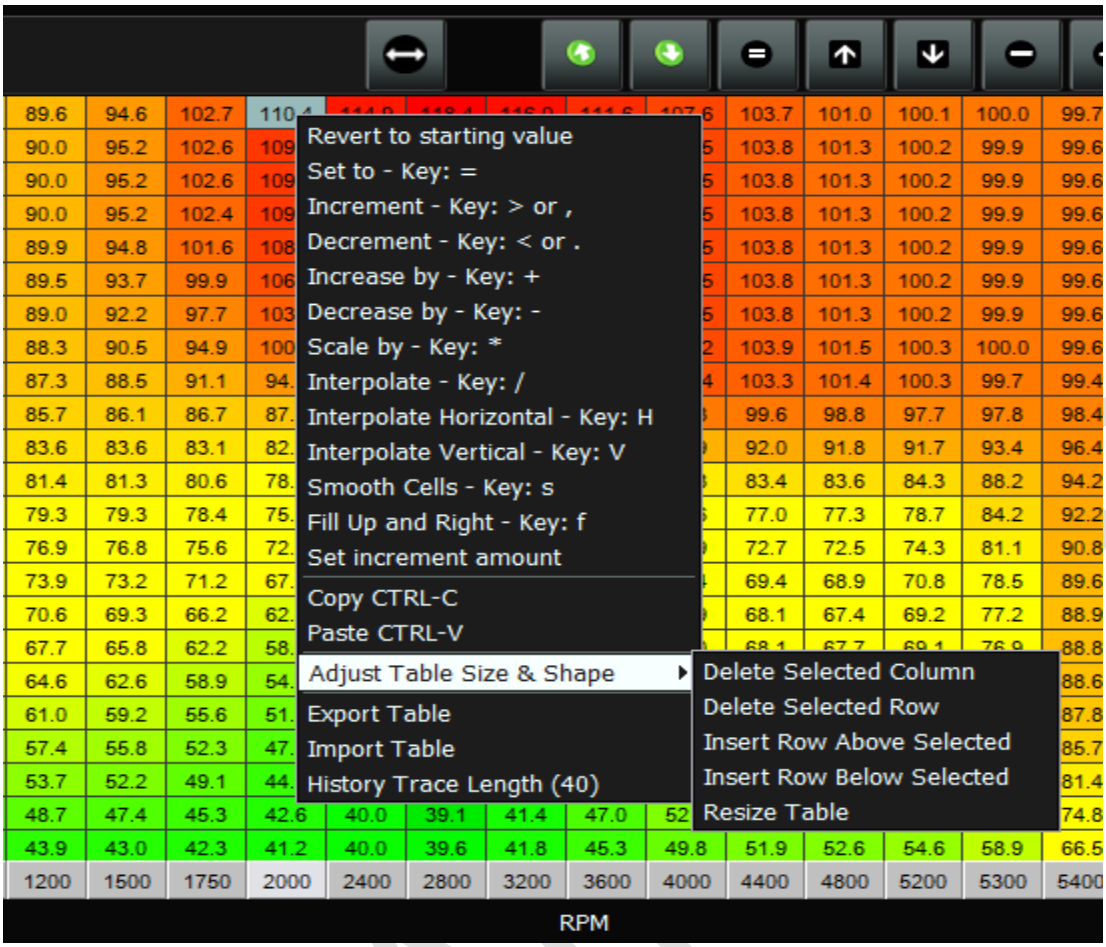
Dynamically sized array Constants can be created for the TableEditor view. To accomplish this, the numeric dimension values are replaced with an expression that will evaluate to the desired dimensions. Consistent with ini expressions, they are to be surrounded with curly brackets. {}

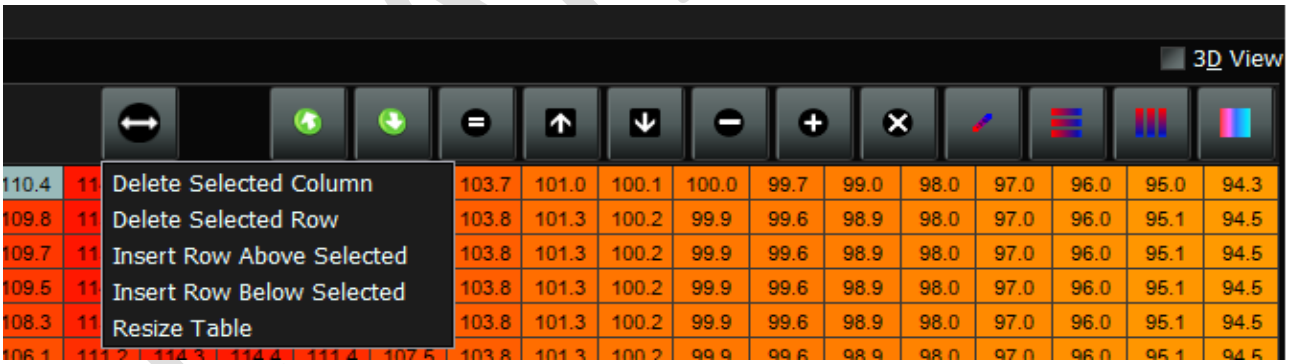
To make dynamically sized tables:

```
[Constants]
   page = 1
   fuel_rows = scalar, U08, 98, "#", 1.0, 0, 12, 24, 0
   fuel_cols = scalar, U08, 99, "#", 1.0, 0, 12, 24, 0
   fuel_rpm  = array, U16, 100, [{fuel_cols}],"rpm",   1, 0,  0, 10000,   0
   fuel_load = array, U16, 148, [{fuel_rows}],"kPa", 0.1, 0, 20.0, 300.0, 1
   veTable1  = array, U16, 196, [{fuel_cols}x{fuel_rows}], "%",0.1, 0, 0.0,200.0, 1
```

If a table is defined in the [TableEditor] section using the above defined constants, the table size will change size based on fuel_rows and fuel_cols Constants and will change dynamically at run-time. The above example will allow any dimension between 12x12 and 24x24. The Constants used for the X, Y and Z axis' will always begin at the defined offset and use only the needed memory, the remainder of the memory will be unreferenced.

When a 2D array has dynamic sizing is used, additional features are unlocked in TunerStudio for changing the size and maintaining the Z axis data.





These re-size menus off the user the ability insert or delete rows or columns from the table as well as a setting a whole new shape for the table. The insert menus are only shown if there is sufficient space remaining in the Z axis array and the Row/Column count are within the min and max limits defined for the X & Y constants. When a row or column is inserted, the initial values will be interpolated from the adjacent cells. If the table is resized, the Z axis will use interpolation to most loosely represent the table of the former size.

In all cases of re-sizing a table, the value of the constant for the row and column count will be updated and the entire new Z axis array will be sent to the controller.

4.4.5.6 string

Allocates a byte range to hold ASCII characters to be used as a String.

```
myStringConstant  = string, ASCII, 0, 20
```

This entry would create a String 20 characters long starting at offset 0.

Optionally, you can over-ride the number of columns to display in the UI rendering:

```
myStringConstant  = string, ASCII, 0, 20, 10
```

4.4.5.7 oddArray

Rarely used read only parameter class. This is used to map memory to an existing single array, but will then act as an array half as large only displaying every other row.

```
Y_Axis_Half    =oddArray,  U08,        80,   [16],  "kPa", 1.0, 0, 0, 255, 0
```

Will create a 1D Array that spans 16 elements, but acts as an 8 element array displaying only every other index. 1,3,5,7,9,11,13,15

4.4.6 Units

Each Constant can be assigned a Units String. This will be referenced throughout the application for display to users on screen and in data logs where applicable.

4.4.7 Scale and Translate

Scale and translate are values used to convert the raw binary values to a user value. Where scale is a multiplier, translate can be used to shift user value.

Applies to all parameter class other than bit and string

Scale and Translate can be fixed numeric values, or expression based.

The scaling and translation values are used as follows:

```
rawValue   = userValue / scale – translate
userValue = (rawValue + translate) * scale
```

Alternatively, if a useScaleAsDivisor ConstantExtention is used and the expression resolves to true:

```
useScaleAsDivisor = constantName, { some expression }
```

then:

```
rawValue   = (scale - (translate*userValue)) / userValue;
userValue = scale / (rawValue   + translate)
```

4.5 Last Attributes Keywords

noMsqSave – Will not load the value from the Calibration file. This can also be defined in the [ConstantsExtension] section and must be for bit type Constants.

controllerPriority – Will save the value to the Calibration file and load it from there if not online with a controller. However, any time a controller is connected, the value from the controller is always silently taken and replaces any value is in the local store.

controllerPriority Constants can only be changed while connected to the controller. Offline tuning can not be performed as any change will be unquestionably taken from the controller on connect.

Example:

```
SERIAL_NUMBER = scalar, U16, nextOffset, "", 1.0, 0, 0, 65535, 0, controllerPriority
```

; The SERIAL_NUMBER is saved for offline knowledge and use, but as soon as we connect to a controller, the serial number of that attached controller will be unquestionably accepted.

5 [PcVariables]

PcVariables – are defined similar to and behave as Constants. The key difference is that PcVarible Values are not stored on the controller, only in the PC memory. They are saved to and loaded from Calibration files.

The definition is the same as a Constant except there is no offset.

PcVariables can be referenced as a Constant in any place a Constant is used.

PcVariables are loaded from saved calibration files before Constants in the order they are defined in the ECU Definition file, thus it is safe to reference PcVariables in Constant expression based attributes.

These are useful to display user preference options or define read only arrays to be used as axis values for [CurveGraph] or [TableEditor] objects.

To use as an axis, define the array in this section, then populate the default values in the [ConstantExtensions] section.

[PcVariables]

```
bst_time_fixed= array, U08, [16],   "ms",   0.32, 0,    0,   5.1,   2, noMsqSave
```

[ConstantExtensions]

```
defaultValue = bst_time_fixed , 0.0 0.32 0.64 0.96 1.28 1.60 1.92 2.24 2.56 2.88 3.20 3.52 3.84 4.16 4.48 4.80
```

Note the noMsqSave flag as this is a read only PcVariable and we do not want to save the value to the Calibration file.

5.1 Specialized PcVariables:

There are 2 specialized PcVariable paramClass types. These allow you to create a PcVariable that represents the value of an OutputChannel. This enables use in enable and visible expressions that can be evaluated during offline tuning when Outputchannel values have not been initialized.

channelValueOnConnect – A specialize paramClass that will retrieve and store the value of an OutputChannel. This will only retrieve the OutputChannel value on connect, then will assume the value does change throughout the communication session. The value will be persisted in the cal file as a PcVariable and be honored for off line tuning.

Syntax:

pcVariableName = channelValueOnConnect, referencedOutputChannel

continuousChannelValue – This will continuously monitor the OutputChannel state for changes during a communication session. The last value during the communication session will be stored in the cal file for offline tuning.

Syntax:

pcVariableName = continuousChannelValue, referencedOutputChannel

6 [CONSTANTEXTENSIONS]

The [ConstantsExtensions] section allows additional attributes and values to be applied to Constants already defined in the [Constants] section.

6.1 KEYWORDS

6.1.1 requiresPowerCycle

Usage:

requiresPowerCycle = constantName

Effect: The constant will be monitored for any user changes. If changed, a notification “Powercycle Required” will be displayed on the dashboard. Normally used with any constant that requires the Controller to be rebooted for the change to take effect.

constantName can be any Constant or PcVariable

6.1.2 defaultValue

Scalar Usage:

defaultValue = rpmhigh, 9000

Effect: sets the initial value of rpmhigh to 9000 If this constant or PcVariable has never been initialized. Once initialized or changed by the user, what ever value they set will be used going forward. However, in a case where the PcVariable or Constant is flagged noMsqSave, this value will always be used.

Array Usage:

defaultValue = tpsBins , 0 8 16 24 32 40 48 56 64 72 80 88 96 100

Provide a space separated list of values

6.1.3 rawValue

Similar to defaultValue except the values are provided as unscaled controller values so scaling can be provided after the fact. This can be useful with expression based scale.

6.1.4 controllerPriority

Usage:

controllerPriority = someConstant

Effect: The value of this constant will always silently accept the controller value. Values will not be loaded from a Calibration file.

6.1.5 readOnly

Usage:

readOnly = constantName

Effect: The Constant or PcVariable is marked as a readOnly entity where the controller value or defaultValue will always drive the value. Values will not be loaded from Calibration Files or editable in the UI.

constantName can be any Constant or PcVariable

6.1.6 reverseIndex

Usage:

reverseIndex = someArrayConstant

Effect: the index of the elements in the array will be access in reverse order from the normal order low to high. Yes some firmwares do order the arrays different between tables.

6.1.7 oppositeEndian

Usage:

oppositeEndian = someConstantName

Effect: The byte order will be treated opposite of the global endianness for this constant. So if the global Endianness is set to big, this constant will be treated with little endianness.

6.1.8 useMappingTable

Usage:

useMappingTable = someConstantName, someMappingFile.inc

Effect: Scale and translate will be driven by the mapping within the inc file. This is used for cases where the scale is not linear such as with temperature sensors.

See more on supported inc files at:

http://www.tunerstudio.com/index.php/manuals/104-supported-inc-file-formats

6.1.9 maintainConstantValue

Usage:

maintainConstantValue = someConstantName, { some expression resolving to the desired value}

Effect: the targetConstant value will always be set to the result of the expression. Each component of the expression is monitored to change to trigger an update to the constant.

6.1.10 maximumElements

Usage: Optional

maximumElements = someConstantName, maximumElements

Where:

someConstantName – A defined Constant of paramClass array with 2 dimensions and size defined by other Constant(s).

maximumElements – An integer value representing the maximum total number of cells to be allowed in a resizable array.

Optional - If not defined, the number of maximum elements will be set to use all available memory to the next defined Constant.

7 [SettingGroups] and Directives

7.1 Overview

Blocks of the ECU Definition can be placed in conditional blocks with if statements. The conditions are limited to flag options that will be set in Project Properties and will require a project reload to take effect. Thus it is preferred that expressions are used in constant and channel definitions over this method as they will take effect immediately and do not require a disruption to the user. However, in some circumstances this approach may be desirable.

7.2 Usage

```
#set MY_CONDITION
#unset MY_OTHER

#if MY_CONDITION
    myDoubleArray     = array , U08,   192,   [12x12], ":1",    0.10000,  0.00000,  1.00,   25.00,    1
#else
    myDoubleArray     = array , U08,   192,   [12x8], ":1",    0.10000,  0.00000,  1.00,   25.00,    1
#endif
```

Using either #set or #unset initializes a flag and it will now appear in the Settings Tab of Project properties. If initialized with #set, the default state will be active, #unset will initialize with a default state of deactivated. In either case the user can over-ride this in Project Properties. In the above example, if MY_CONDITION is active, the array will be defined as a 12x12, otherwise it will be a 12x8 array.

#set, #unset, #if, #else, #elif and #endif can be used anywhere in the ECU Definition, they are not limited to [SettingGroups].

What is limited to [SettingGroups] is the definition of multi-choice options.

```
[SettingGroups]
    settingGroup = lambdaSensor, "Oxygen Sensor / Display"
    settingOption = NARROW_BAND_EGO, "Narrowband Sensor - Volts"
    settingOption = LAMBDA, "Wideband - Lambda"
    settingOption = DEFAULT, "WideBand - AFR" ; DEFAULT will be over looked and this
                                ; will fall into the #else block of the statement.
```

This will now be presented in Project properties as 1 dropdown with 3 options. In your ECU Definition you can then change the definition depending on the user selection.

7.2.1 Example

```
Example OutputChannel definition dependent on selection:

#if NARROW_BAND_EGO
   afrtgt1          = scalar, U08,   12, "Volts", 0.00489, 0.0
#elif LAMBDA
   afrtgt1raw       = scalar, U08,   12, "Lambda", 0.1, 0.0
#else
```

```
   afrtgt1          = scalar, U08,   12, "AFR", 0.1, 0.0
#endif
```

7.3 ADDITIONAL DIRECTIVES

7.3.1 #define

#define allows you to define string lists with over-rides as a short cut for redundant bit field definitions. These lists can also be created by referencing other #define lists as fragments to create a larger list.

7.3.1.1 Example:

A bit constant defined with the standard notation:

```
als_out_pin    = bits,    U08,    344, [0:2], "Off", "IAC1", "IAC2", "FIDLE"
```

Can be replaced with:

```
; the define would typically be at the top of the [Constants]
; but can be anywhere in the ECU Definitions file
#define PINLIST = "Off", "IAC1", "IAC2", "FIDLE"

; this is also supported to break into smaller lists:
#define LOWLIST = "Off", "IAC1"
#define MIDLIST = "IAC2"
#define PINLIST = $LOWLIST, $MIDLIST, "FIDLE"

; In both of the above, PINLIST would resolve to the same end result.

; the Constant definition would remain where it is, but reference the PINLIST
als_out_pin    = bits,    U08,    344, [0:2], $PINLIST
```

Or if you want to use the list, but change 1 or more items in the list:

```
; the define would typically be at the top of the [Constants]
; but can be anywhere in the ECU Definitions file
#define PINLIST = "Off", "IAC1", "IAC2", "FIDLE"

; the Constant definition would remain where it is, but reference the PINLIST
als_out_pin    = bits,    U08,    344, [0:2], $PINLIST, 1=”INVALID”, 2=”STEPPER”
```

The applied over-rides will result in the equivalent of index 1 being flagged INVALID and 2 changing names from IAC2 to STEPPER.

7.3.2 #include

The #include will direct the parser, to load the additional file as part of the current ECU Definition

Usage:

#include myOtherFile.ini

The application will check for the file in the folder:

[ProjectFolder]/projectCfg/

If no found there it will check in the application install.

8 [CONTROLLERCOMMANDS]

8.1 GENERAL

Controller commands are used to define instructions made up of 1 or more bytes to trigger an action by the controller.

```
commandName     = command1, command2, commandn...
```

command in standard ini format, a command name can be assigned to 1 to n commands that will be executed in order.

This does not include any resultant protocol envelope data, only the response data itself.

WARNING!! These commands bypass TunerStudio's normal memory synchronization. If these commands alter mapped settings (Constant) memory in the controller, TunerStudio will have an out of sync condition and may create error messages.

It is expected that these commands would not typically alter any ram mapped to a Constant.

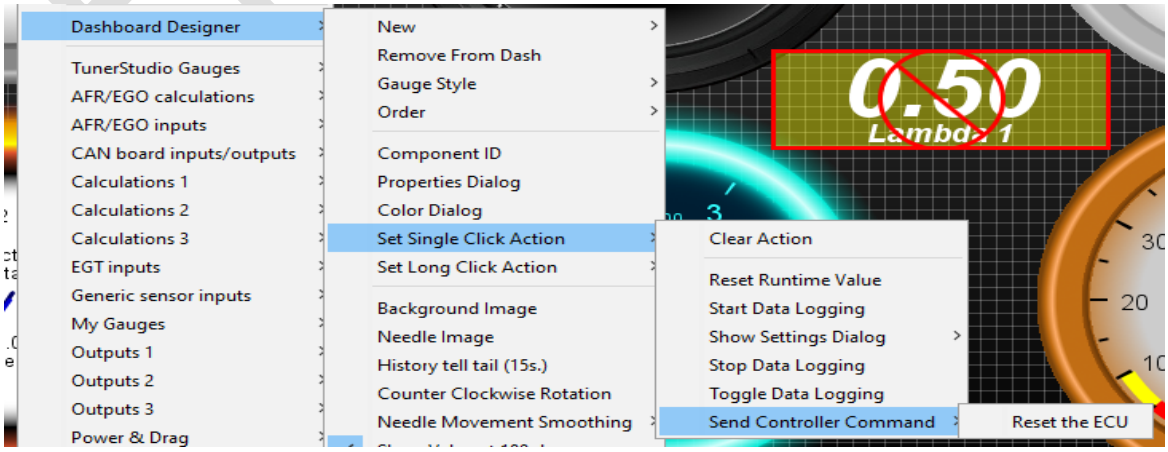
```
cmdReset = "xf1\x02\x87\x00"
```

controllerCommands can then be assigned to a Button in a dialog or assigned to a touch action on a dashboard to be triggered and sent.

8.2 USER DISPLAYED CONTROLLER COMMANDS

By default, Controller Commands defined in the ini are for use only in the ini file and not visible for user assignment. The reasoning for this is that these Commands can be powerful and the firmware developer will likely want to maintain control over what ones are user actions. However, the ini developer can enable specified ControllerCommands for user assignment. For instance, a ControllerCommand can be assigned for single or double touch triggering from any Gauge or Indicator.
To enable the above cmdReset for user assignement, use this entry in the ControllerCommands section:

```
displayCommand = cmdReset, "Reset the ECU"
```



9 [CURVEEDITOR]

9.1 1D ARRAY GRAPH EDITORS

The [CurveEditor] section allows you to define graph views for editing 1D arrays. 1 to N 1D arrays can be configured in a single view. A minimum of 2 – 1D arrays are required to define a Curve Graph, 1 X axis array and a Y Axis array. You may add as many additional 1D arrays to the Y axis as desired, but the Y axis min and max will be the same for all Y axis arrays. All arrays defined on a graph must be of the same length. For tracking of runtime data on a curve graph, an OuputChannel matching the units of the X Axis array must be set, this will keep the green bubble tracking to the X axis and the intersection on the 1st Y Axis Array. An additional OutputChannel can be set for the Y Axis in which case the bubble will track to the 2 channels intersection, this is often desirable with multiple Y axis arrays on a single graph.

The definition of a Curve Editor is a multi-line definition that can relatively basic or rather complex depending on the need.

9.2 ENTRY SYNTAX

9.2.1 Required Entries

```
curve = curveEditorName, "Title"
     columnLabel = "X Axis Label", "Y Axis Label"
     xAxis      =  xMin,   xMax, numVerticalDivisions
     yAxis      =  yMin,   yMax, numHorizontalDivisions
     xBins      =  xAxisArray, xChannel
     yBins      =  yAxisArray, yChannel
```

curveEditorName – A name you assign to this curve editor to reference from other sections. This an alpha-numeric name that must start with a letter and not use mathematical operators

columnLabel – The labels to be displayed on the X and Y Axis, units of the Constant will also be appended. If units are based on a String Function, they will dynamically be updated.

xAxis – Defines the min and max of the X Axis. These values can be numeric or an expression enclosed in curly brackets { }. numVerticalDivisions will define the number of light grey lines painted vertically.

yAxis – Defines the min and max of the Y Axis. These values can be numeric or an expression enclosed in curly brackets { }. numHorizontalDivisions will define the number of light grey lines painted horizontally.

xBins – Defines the X Axis Array (required), OutputChannel to track to (Optional) and a readOnly flag can optionally be appended if you do not wish the x array values be edited in this view.

yBins – Defines the Y Axis Array (required), OutputChannel to track to (Optional, if used xChannel is required). One yBins row is required, but multiples can be used by adding additional rows with other 1D array references. In multi Y Axis Curve Editors, an expression parameter can be appended that evaluates to whether that y array is active or not.

9.2.2 Optional Attributes

```
topicHelp = "file://$getProjectsDirPath()/docs/HelpDoc_x.x.pdf#xxx"
showTextValues = true ; Will show text inputs by default
lineLabel   = "Y Axis Label" ;
size        = 480, 350 ; Suggested width, height
gauge       = someGauge
showXYDataPlot = true, xAxisLogField, yAxisLogField, { expressionPosition }
```

topicHelp – A optional entry that will enable a help menu that deep links into the document. helpManualDownloadRoot should be defined in the [TunerStudio] section

showTextValues – If set to true, the text cells will be visible by default, otherwise they will not be visible until the user toggles them on. The default behavior is false

lineLabel – Will place a color coded label along the X axis. This is useful with multiple Y Axis Arrays, include a lineLabel for each in the same order the Y Axis arrays were added.

size – Alter the default sizing of the dialog. Support for this is up to the application and platform.

gauge – By adding a reference to a gauge defined in [GaugeConfigurations], this gauge will be displayed in the top right corner and the text cells will be laid out vertically under the gauge.

showXYDataPlot – This activates an X-Y plot but default using the field names provided. The expression defines the start of the X Axis plot.

Example:

```
showXYDataPlot = true, Time, TOSS_rpm, { Tmr_Enable > 0 }
```

This entry will activate an X-Y plot on the Curve Editor with the X Axis field being Time, Y Axis Field of TOSS-rpm, the start point of the X Axis will be where Tmr_Enable becomes a non-zero value.

suppressGraph – By Default false. When set to true, the text entries will be displayed without the graph.

9.3 Example Curve Editors

9.3.1 Basic single 1D Array Curve Editor

1 X Axis array is always required, but can be set as a read only reference. In a case where the X Axis values are hard values known to the controller, but not within the calibration data; you would create a read only PcVariable array with default values matching those known to the controller.

9.3.1.1 View:

Basic 1D Array Editor – 1 Editable Array

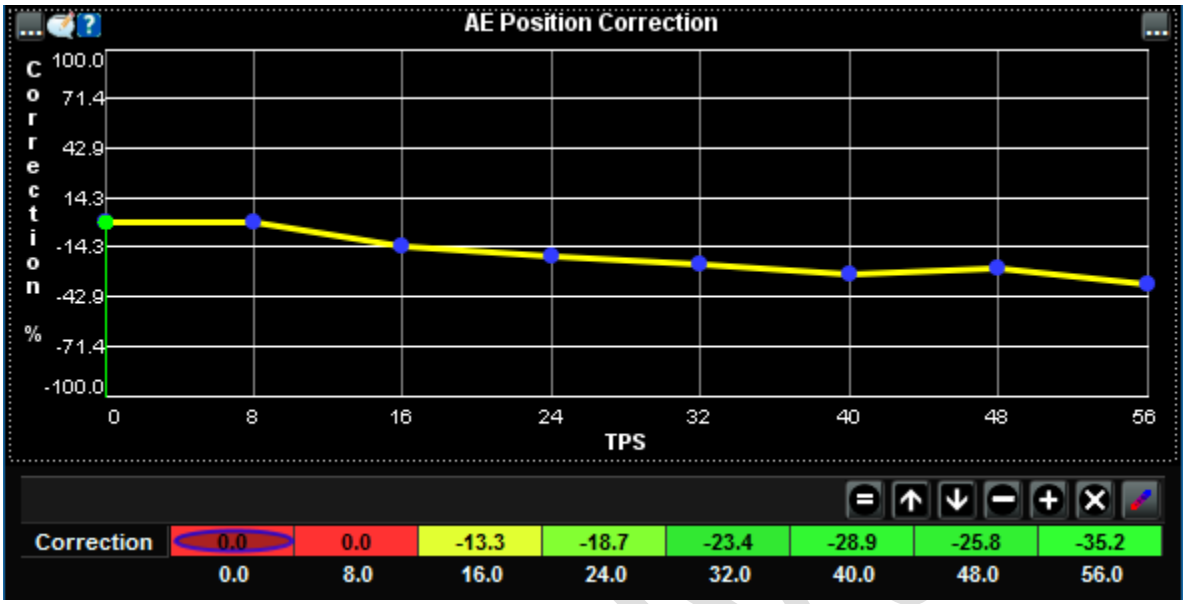


Figure 9-1

9.3.1.2 Definition:

```
curve = ae_posCor, "AE Position Correction"
     columnLabel = "TPS", "Correction"
     xAxis       =     0,  56, 8
     yAxis       =  -100, 100, 8
     xBins       =  AE_tpsPer, TPS_Pct, readOnly
     yBins       = AE_Pos_Table
     showTextValues = true
```

9.3.2 Curve Editor with 2 editable arrays and a gauge

9.3.2.1 View:

1D Array Editor – 2 Editable Arrays with Gauge

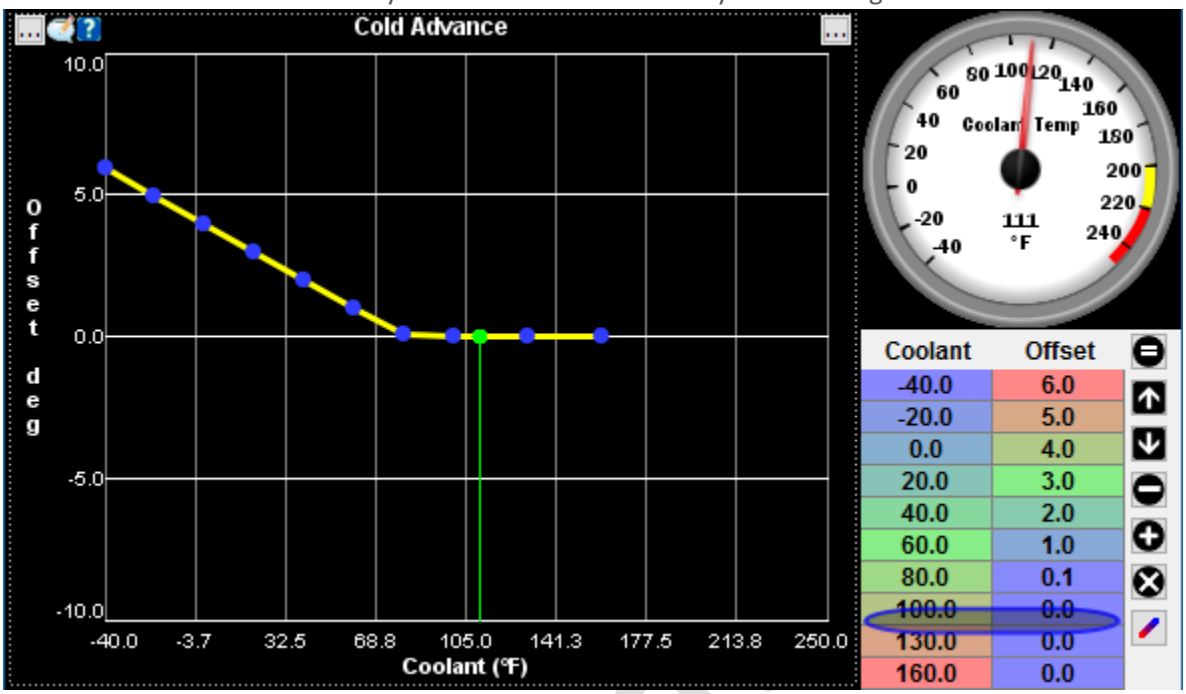


Figure 9-2

9.3.2.2 Definition:

```
curve = coldAdvance, "Cold Advance"
   topicHelp = "file://$getProjectsDirPath()/docs/HelpDoc_x.x.pdf#coldadv"
   columnLabel = "Coolant", "Offset"
   xAxis       = -40, {clthighlim}, 9
   yAxis       = -10,  10, 5
   xBins       = tempTable, coolant, readonly
   yBins       = cold_adv_table
   gauge       = cltGauge
```

9.3.3 Complex Curve Editor – Multi-Conditional Arrays

9.3.3.1 View:

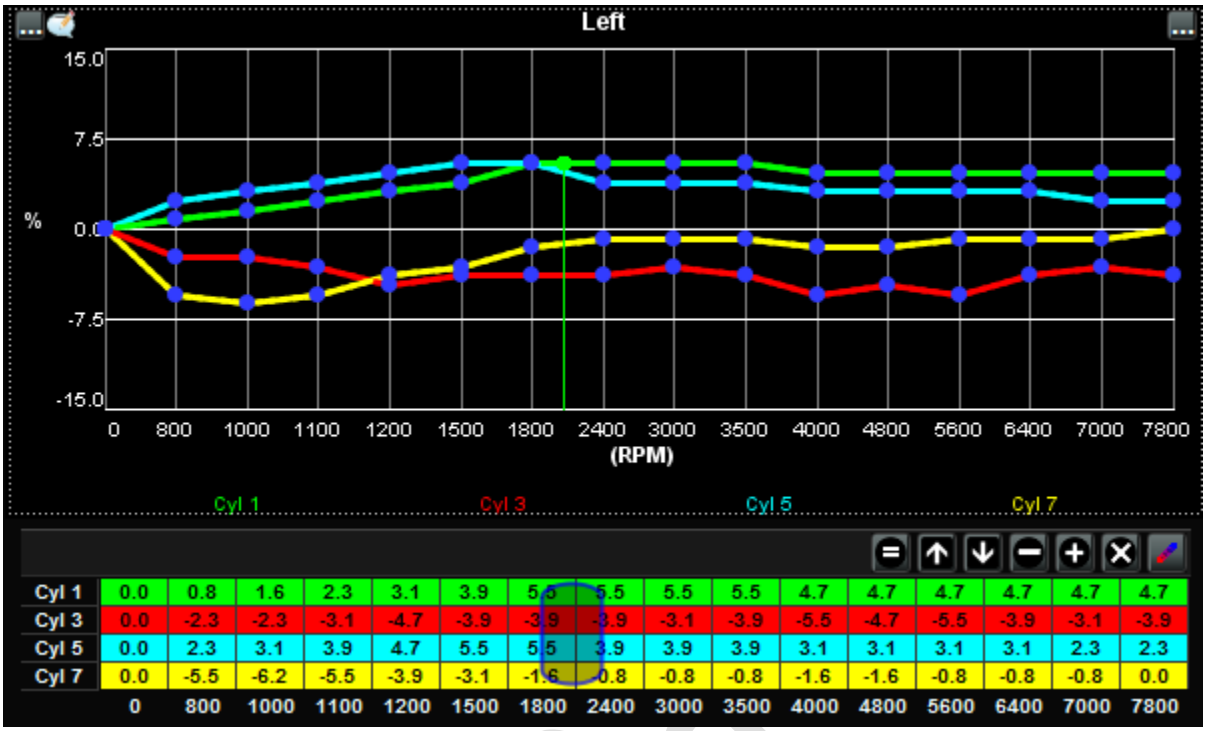


Figure 9-3

9.3.3.2 Definition:

In this example we have 4-8 cylinders, but only want to display the left bank on this Curve editor. However, the cylinder order is set in another constant. So this Curve Editor actually has 8 yAxis arrays assigned to it; each with an expression to determine which arrays to actually present to the user. The expressions are constructed so that it will resolve to true if the number is odd and less or equal than the maximum cylinders. Another similar Curve Editor was constructed for the Right bank.

```
curve = cyl_Trim_Left, " Left "
   columnLabel = "", ""
   xAxis       =   600, {rpmhigh}, 16
   yAxis       = -15, 15, 5
   xBins       = X_Axis_Table, RPM, readOnly
   yBins       = ICF_Table_a, { engType == 0 ? Fire_Order_a <= cylCount / 2 : Fire_Order_a % 2 == 1 }
   yBins       = ICF_Table_b, { engType == 0 ? Fire_Order_b <= cylCount / 2 : Fire_Order_b % 2 == 1 }
   yBins       = ICF_Table_c, { engType == 0 ? Fire_Order_c <= cylCount / 2 : Fire_Order_c % 2 == 1 && num_of_cyl >= 0 }
   yBins       = ICF_Table_d, { engType == 0 ? Fire_Order_d <= cylCount / 2 : Fire_Order_d % 2 == 1 && num_of_cyl >= 0 }
   yBins       = ICF_Table_e, { engType == 0 ? (cylCount < 6 ? 0 : FireOrder_e <= cylCount /2 ): FireOrder_e % 2 == 1 && num_of_cyl >= 1 }
   yBins       = ICF_Table_f, { engType == 0 ? (cylCount < 6 ? 0 : FireOrder_f <= cylCount /2 ): FireOrder_f % 2 == 1 && num_of_cyl >= 1 }
   yBins       = ICF_Table_g, { engType == 0 ? (cylCount < 8 ? 0 : FireOrder_g <= cylCount /2 ): FireOrder_g % 2 == 1 && num_of_cyl >= 2 }
   yBins       = ICF_Table_h, { engType == 0 ? (cylCount < 8 ? 0 : FireOrder_h <= cylCount /2 ): FireOrder_h % 2 == 1 && num_of_cyl >= 2 }
   ;gauge       = afr1Gauge
   lineLabel   = "Cyl $stringValue( Fire_Order_a)"
   lineLabel   = "Cyl $stringValue( Fire_Order_b)"
   lineLabel   = "Cyl $stringValue( Fire_Order_c)"
   lineLabel   = "Cyl $stringValue( Fire_Order_d)"
   lineLabel   = "Cyl $stringValue( Fire_Order_e)"
   lineLabel   = "Cyl $stringValue( Fire_Order_f)"
   lineLabel   = "Cyl $stringValue( Fire_Order_g)"
   lineLabel   = "Cyl $stringValue( Fire_Order_h)"
   showTextValues = true
   size        = 500, 350
```

9.3.4 Curve Editor with X-Y Plot Active

9.3.4.1 View:

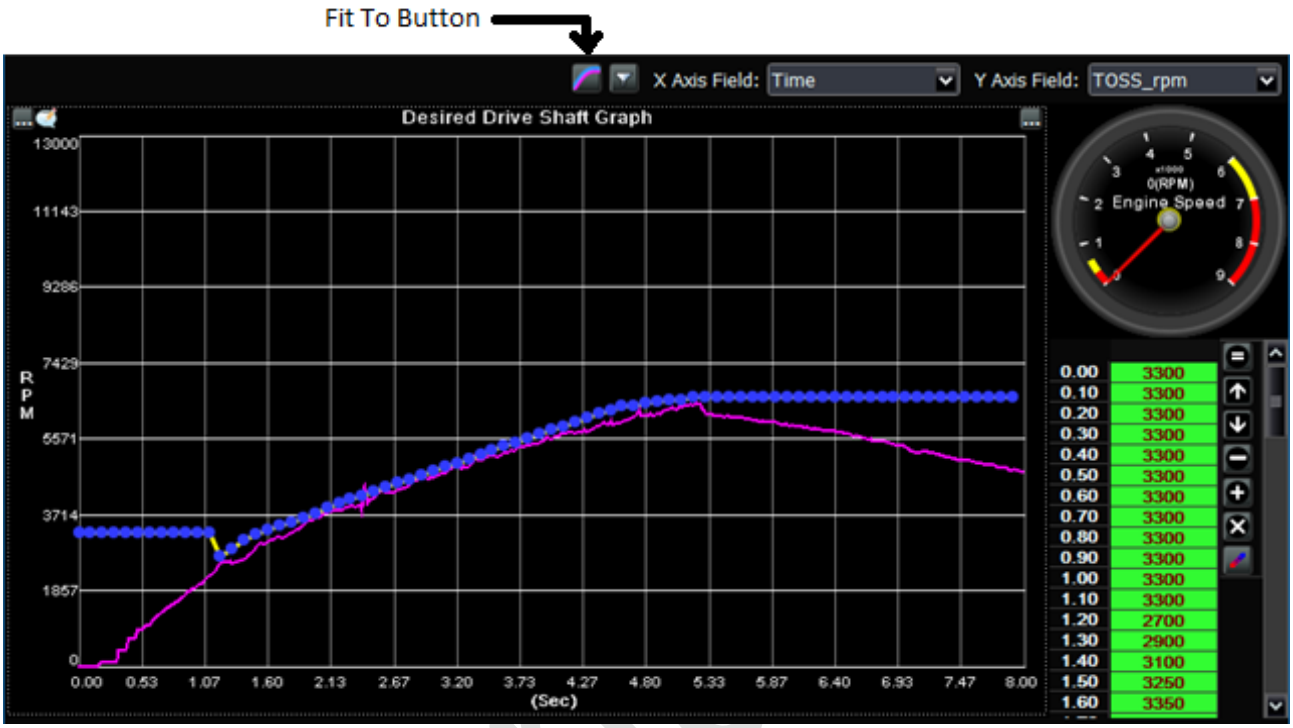


Figure 9-4

9.3.4.2 Fit To

The “Fit To” Button will fit all selected points on the Curve Editor to the Data Logs X-Y plot values.

9.3.4.3 Definition:

```
curve = des_ds_Tab, "Desired Drive Shaft Graph"
    columnLabel = "", ""
    xAxis       =  0.0,   8.0, 16
    yAxis       =    0, 13000, 8
    xBins       = dsr_ds_grh, Time, readOnly
    yBins       = Dsrd_DS_Table, TOSS_rpm
    gauge       = Tachometer
    showTextValues = true
    showXYDataPlot = true, Time, TOSS_rpm, { Tmr_Enable > 0 }
```

10 [TableEditor]

10.1 Overview

The [TableEditor] section is used to define the relationship between 2 - 1D array constants and 1 - 2D Arrays to produce a visual editor using 2D or 2D views.

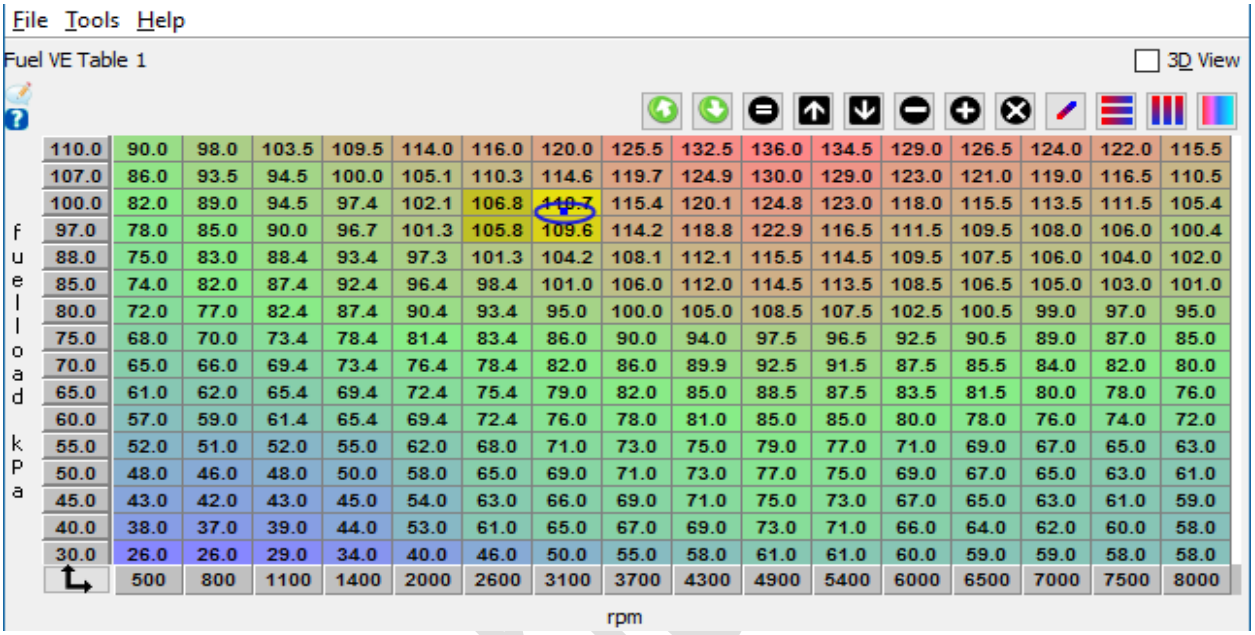


Figure 10-1

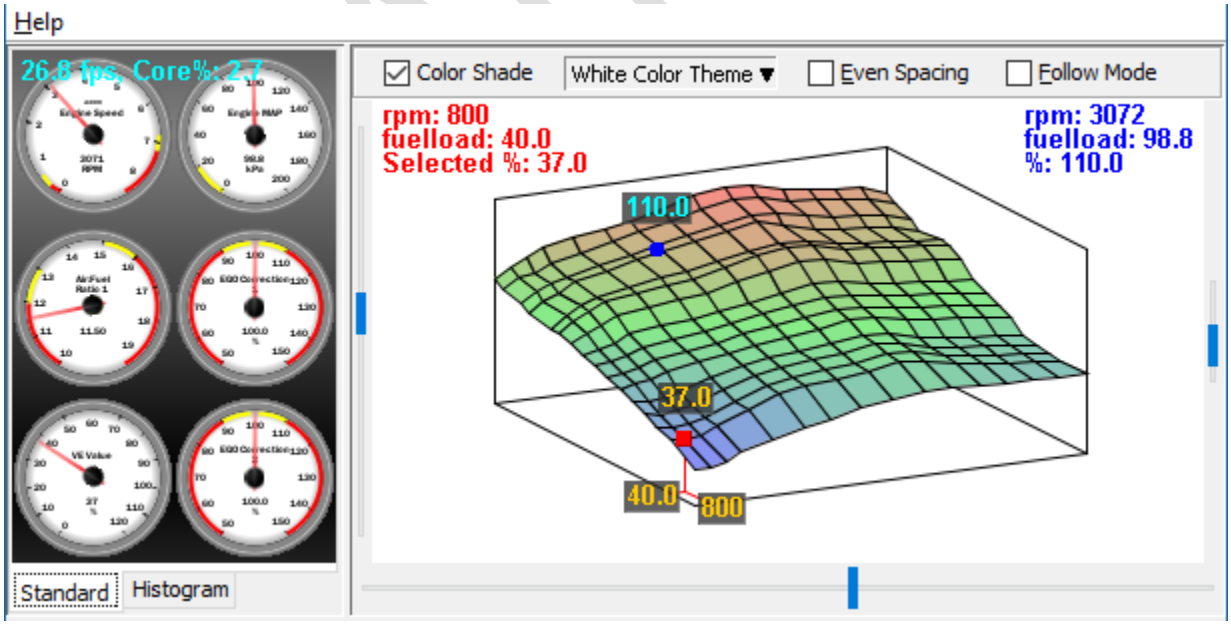


Figure 10-2

The size of the 1D array constants used on the x & y axis must match the dimensions of the 2D array. The entry to define a table is a multi-line entry as follows:

10.2 SYNTAX:

```
;table = table_id,         map3d_id,          "title"
 table = veTable1Tbl,      veTable1Map,      "VE Table 1",    2
   topicHelp = "file://$getProjectsDirPath()/docs/HelpFile1.2.pdf#someAnch"
   ;              constant,   OutputChannel
   xyLabels     = "X Label", "y Label"
   xBins        = some1Darray, channelTrackingArray
   yBins        = some1Darray, channelTrackingArray
   zBins        = some2Darray

  upDownLabel = "UP Label", "Down Label" ; labels for the 3D view
  gridOrient  = 250,   0, 340 ; Space 123 rotation of grid in degrees.
```

gridOrient – defines the default display angle in the 3D view. This is an optional entry

topicHelp – A optional entry that will enable a help menu that deep links into the document. helpManualDownloadRoot should be defined in the [TunerStudio] section

xyLabels – [Optional] Set the X & Y Label to be displayed on the table. This can be a String or String Function. The units of the underlying Constant will be appended. If the xyLabel is not defined, the labels and units for the X & Y axis will be pulled from the underlying Constants. If the units are based on a StringFunction, they will be updated dynamically on when conditions change.

Once defined a table can be access by being set as the target of a Menu or embedded in a dialog.

To display as a 3D view with a gauge cluster, use the map3d_id, for a standard 2D view use the table_id as the target.

10.2.1 Example Entry:

```
;table_id,         map3d_id,          "title"
 table = veTable1Tbl,      veTable1Map,      "VE Table 1",    2
   topicHelp = "file://$getProjectsDirPath()/docs/HelpFile1.2.pdf#someAnch"
   ;              constant,   OutputChannel
   xBins        = frpm_table, rpm
   yBins        = fmap_table, fuelLoad
   zBins        = veTable1

  upDownLabel = "RICHER", "LEANER"
  gridHeight  = 2.0
  gridOrient  = 250,   0, 340 ; Space 123 rotation of grid in degrees.
```

There is an optional readOnly flag that can be placed at the end of the xBins or yBins rows if you wish to prevent users from editing in any particular view.

```
 table = veTable1Tbl,      veTable1Map,      "VE Table 1",    2
   ;              constant,   OutputChannel
   xBins        = frpm_table, rpm, readOnly
   yBins        = fmap_table, fuelLoad, readOnly
   zBins        = veTable1
```

10.3 Resizable Tables

Dynamically sized tables are no different in the TableEditor section, but the underlying Constants are sized based on expressions as described in 4.4.5.5.1

When a Table is configured to be resizable, the UI will manage the table values for any applied new size.

On any resize event, the values are captured, the table resized, the new values are applied and sent to the box.

In the table widget, the following resize actions are available:

- Insert Column Before Selected
- Insert Column After Selected
- Insert Row Above Selected
- Insert Row Below Selected
- Delete Selected Row
- Delete selected Column
- Resize Table - Set the table to a specific number of rows and columns. The current table will then be interpolated to this new shape.

Each of these options are available if currently valid.

This feature requires ini spec 3.53 is supported by the application.

Here is what needs to be changed in the ini:

```
; Create to new scalar Constants that will represent the array dimensions.
; These must be defined anywhere above the arrays in the ini file.
; The min and max for these constants will drive the minimum and maximum allowed rows and columns.
      FUEL_TABLE_ROWS         = scalar,   U08,  0,     "",   1.0,       0,  8.0,  32.0,    0
      FUEL_TABLE_COLS         = scalar,   U08,  1,     "",   1.0,       0,  8.0,  32.0,    0

; use the Dimension Constants to set the array size.
      FUEL_RPM_AXIS = array,    U16, 2,   [{FUEL_TABLE_COLS}],  "",   1,  0,   0,  15000, 0
      FUEL_LOAD_AXIS = array, U16, 66,    [{FUEL_TABLE_ROWS}],   "", 0.1,  0, 20.0, 1000.0,  1
      veTable1 = array,  U16, 130, [{FUEL_TABLE_COLS}x{FUEL_TABLE_ROWS}], "%", 0.1, 0,0.0, 200.0, 1
```

The application will detect that these tables are sized based on the constants and enabled the resize actions in the widget. The dimensions of the arrays should be a single constant for the resizing tools to be enabled.

Default values for FUEL_TABLE_ROWS and FUEL_TABLE_COLS should be defined in the [ConstantsExtensions] section as demonstrated below. This is required for the table to have a valid size prior to loading a saved calibration or connecting to an ECU.

The min and max number of rows will be driven by the min and max of FUEL_TABLE_ROWS as well as maximumElements for the Z Axis.

The min and max number of columns will be driven by the min and max of FUEL_TABLE_COLS as well as maximumElements for the Z Axis.

The start address / offset for each of the Constants will remain constant, but the size will vary based on the values of FUEL_TABLE_COLS and FUEL_TABLE_ROWS.

This can be configured to alter the start upon request.

By default, as maximumElements for the Z Axis will be determined based on the available memory before the next defined Constant. For example, veTable 1 has a starting address of 130 on the page, if the next Constant on that page has an offset of 706, 576 will be used as the maximumElements.

The maximumElements can be over-ridden by creating an entry in the [ConstantsExtension] section:

```
[ConstantsExtensions]
   maximumElements = veTable1, 576
   defaultValue = FUEL_TABLE_ROWS, 16 ; the default number of rows used for VE Table 1.
   defaultValue = FUEL_TABLE_COLS, 16 ; the default number of columns used for VE Table 1.
```

In this example the maximumElements is defined as 576, this is enough for a 24x24 table in a typical configuration. However as the min and max for FUEL_TABLE_ROWS and FUEL_TABLE_COLS are set to 8 and 32, the application user will be able to resize a table to any dimension between 8x32 to 32x8 where:

FUEL_TABLE_ROWS * FUEL_TABLE_COLS < 576

Any of the following Row, Column pairs will be allowed by the application.

	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
8	64	72	80	88	96	104	112	120	128	136	144	152	160	168	176	184	192	200	208	216	224	232	240	248	256
9	72	81	90	99	108	117	126	135	144	153	162	171	180	189	198	207	216	225	234	243	252	261	270	279	288
10	80	90	100	110	120	130	140	150	160	170	180	190	200	210	220	230	240	250	260	270	280	290	300	310	320
11	88	99	110	121	132	143	154	165	176	187	198	209	220	231	242	253	264	275	286	297	308	319	330	341	352
12	96	108	120	132	144	156	168	180	192	204	216	228	240	252	264	276	288	300	312	324	336	348	360	372	384
13	104	117	130	143	156	169	182	195	208	221	234	247	260	273	286	299	312	325	338	351	364	377	390	403	416
14	112	126	140	154	168	182	196	210	224	238	252	266	280	294	308	322	336	350	364	378	392	406	420	434	448
15	120	135	150	165	180	195	210	225	240	255	270	285	300	315	330	345	360	375	390	405	420	435	450	465	480
16	128	144	160	176	192	208	224	240	256	272	288	304	320	336	352	368	384	400	416	432	448	464	480	496	512
17	136	153	170	187	204	221	238	255	272	289	306	323	340	357	374	391	408	425	442	459	476	493	510	527	544
18	144	162	180	198	216	234	252	270	288	306	324	342	360	378	396	414	432	450	468	486	504	522	540	558	576
19	152	171	190	209	228	247	266	285	304	323	342	361	380	399	418	437	456	475	494	513	532	551	570	589	608
20	160	180	200	220	240	260	280	300	320	340	360	380	400	420	440	460	480	500	520	540	560	580	600	620	640
21	168	189	210	231	252	273	294	315	336	357	378	399	420	441	462	483	504	525	546	567	588	609	630	651	672
22	176	198	220	242	264	286	308	330	352	374	396	418	440	462	484	506	528	550	572	594	616	638	660	682	704
23	184	207	230	253	276	299	322	345	368	391	414	437	460	483	506	529	552	575	598	621	644	667	690	713	736
24	192	216	240	264	288	312	336	360	384	408	432	456	480	504	528	552	576	600	624	648	672	696	720	744	768
25	200	225	250	275	300	325	350	375	400	425	450	475	500	525	550	575	600	625	650	675	700	725	750	775	800
26	208	234	260	286	312	338	364	390	416	442	468	494	520	546	572	598	624	650	676	702	728	754	780	806	832
27	216	243	270	297	324	351	378	405	432	459	486	513	540	567	594	621	648	675	702	729	756	783	810	837	864
28	224	252	280	308	336	364	392	420	448	476	504	532	560	588	616	644	672	700	728	756	784	812	840	868	896
29	232	261	290	319	348	377	406	435	464	493	522	551	580	609	638	667	696	725	754	783	812	841	870	899	928
30	240	270	300	330	360	390	420	450	480	510	540	570	600	630	660	690	720	750	780	810	840	870	900	930	960
31	248	279	310	341	372	403	434	465	496	527	558	589	620	651	682	713	744	775	806	837	868	899	930	961	992
32	256	288	320	352	384	416	448	480	512	544	576	608	640	672	704	736	768	800	832	864	896	928	960	992	1024

11 [OUTPUTCHANNELS]

11.1 OVERVIEW

OutputChannels have two basic forms:

1. Controller Channels - values contained in the runtime data stream received from the controller.
2. Expression based Channels - new channels based on any other channels or Constant using mathematical operations and TunerStudio functions.

Type one are primarily used by Firmware developers using the basic protocols, type 2 allows you to create variables that can be referenced throughout the ECU Definition file and can easily be user extensions.

11.1.1 Type 1 format for scalar:

```
channelName      = scalar, dataType, offset, "Units", scale, translate
```

channelName can be any alphanumeric string. It must start with a letter and contain no special characters or white spaces.

dataType will be U08, S08, U16, S16, U16, S32 or F32. For F32

offset - the index of the 1st byte in the read datastream, this can be numeric or key words nextOffset and lastOffset

Units can be a string or use String function expressions

scale and translate will be applied to the raw value using the standard formulas:

rawValue = userValue / scale - translate

userValue = (rawValue + translate) * scale

Scale and translate can be expressions

Example:

```
seconds       = scalar, U16,    0, "s",   1.000, 0.0
```

11.1.2 Type 1 format for bit

channelName = bits, dataType, offset, bitsOfInterest

```
Examples:
  ready             = bits,   U08,   11, [0:0]
  crank             = bits,   U08,   11, [1:1]
  startw            = bits,   U08,   11, [2:2]
  warmup            = bits,   U08,   11, [3:3]
  tpsaccaen         = bits,   U08,   11, [4:4]
  tpsaccden         = bits,   U08,   11, [5:5]
```

6 bit fields defined from 1 byte at offset 11.
the bits of interest are described in the format [n:m] where n is the starting bit and m the last bit.
in the above examples, n=m so each channel is a single bit.

11.1.3 Type 2 format – Expression based Channels:

channelName = { someExpression }, "Units"

someExpression can be made up of any set of Constants, PcVariables and OutputChannels using any of the TunerStudio operators and functions.

11.1.3.1 Example formula based Channels:

```
[OutputChannels]

vacuum = {(barometer-map)*0.2953007}, "inHg" ; Calculate vacuum in in-Hg.
boost = {map < barometer ? 0.0 : (map-barometer)*0.1450377}, "PSI" ; boost in PSIG.
```

When using complex expressions as scale, translate or limit, it is generally preferred to create an expression based OutputChannel that implements the complex components, then you can use that single channel as a variable

For more information on functions, see the **Expressions and Functions** section of this document

11.2 OPTIMIZED OUTPUTCHANNEL CAPTURE

There are 3 ways this is read, essentially they all use the equivalent of an UPLOAD command..

1) Un-optimized

2) Optimized blocking

3) Optimized – High Speed

11.2.1 Supporting Un-Optimized

The entire block from the start address to the highest defined channel is read using the largest block sizes supported by the firmware.

To support un-optimized reading, 2 attributes should be set:

ochGetCommand – the Command that will be issued to read the defined output channel block;

ochBlockSize – the number of bytes to be expected when the command is issued.

Example:

```
ochBlockSize     =    112 ; 112 bytes will be returns when the A command is issued
ochGetCommand       = "A"
```

11.2.2 Optimized Blocking

The master always tracks what channels are currently being used at any point in time by anything (Dashboards, data logging, UI widgets, etc..). It will use this info to break the reads of channels into blocks. Thus it may read 3 block of sizes 40, 15, 3 in 3 UPLOADS instead of the entire defined block. There is a hysteresis where X number of unused bytes exist between used bytes to account for the additional request time.

To support Optimized blocking, the ochGetCommand must have the tags for offset and length and the firmware must support this as a read command of varying start positions and lengths.

Example:

```
ochGetCommand    = "r\$tsCanId\x07%2o%2c"
```

At run-time the tags for offset “%2o” will be replaced with 2 bytes to represent the offset or address.
At run-time the tags for offset “%2c” will be replaced with 2 bytes to represent number of bytes to be read.

11.2.3 Full Optimized – High Speed

With full optimized; the master maintains an ordered table on the slave of offsets and lengths that are currently being used. A command from master to slave is issues where the slave returns an ordered byte array that only contains a subset of the data at the based on the offset and lengths set in a 1D array on the controller. So no more data is returned than is currently needed. The master is then responsible for tracking what channels are at each position in the run-time subset.

To support Full Optimized, additional attributes must be set and supported.

scatteredOffsetArray – this must point to a U16 Constant of paramClass array. The array size will drive the maximum number of channels that can be read by this means. If it should overflow, the master will fall back to run-time reads using 1 of the previous methods. Each U16 is populated in the following manor: top 3 bits state data size in bytes, lower 13 bits contain the offset of the channel.

Size2	Size1	Size0	Off12	Off11	Off10	Off9	Off8	Off7	Off6	Off5	Off4	Off3	Off2	Off1	Off0

Data size chart:

1 = BYTE
2 = WORD
3 = DWORD
4 = QWORD

scatteredOchGetCommand – This defines the command to be sent to retrieve the runtime subset.

scatteredGetEnabled – an expression that will activate / deactivate high speed optimized run-time

```
scatteredGetEnabled =  { scatterRuntimeEnabled && (tsLocalCanId == tsCanId) }
```

Example ini entries:

```
[Constants]
…
…
page = 16
      qfrtfielddata= array,   U16, 0, [256], "", 1.0000, 0.00000,0.00,65535, 0, noMsqSave

[OutputChannels]
   ochBlockSize     = 512 ; change this if adding extra data to outpc
   ochGetCommand    = "r\$tsCanId\x07%2o%2c" ; leave this alone

   scatteredOffsetArray = qfrtfielddata
   scatteredOchGetCommand = "g"
   scatteredGetEnabled =  { scatterRuntimeEnabled && (tsLocalCanId == tsCanId) }
```

Once you define the scatteredOffsetArray in the ini, TunerStudio will maintain the values in that based on what is actually being used. Inside TS OutputChannels are managed by a pub/sub engine. TS knows if there is anything subscribed to every channel.

Whenever anything subscribes or unsubscribes to a channel, TS will update the values in the scatteredOffsetArray.

When the scatteredOchGetCommand is issued, TS will be expecting a response containing what is defined in the scatteredOffsetArray.

For Example:
  scatteredOchGetCommand = "g"
  scatteredOffsetArray = offsetArray;

TS only has 2 channels subscribed, 1 16 bit at offset 22, 1 32 bit channel at offset 48.
As the top 3 bits of the U16 is size in bytes, the lower 13 bits are the offset.

TS will write to offsetArray:

offsetArray[0] = 0xA016
offsetArray[1] = 0xC048
offsetArray[2-n] = 0x0000 – this indicates end of chunks, no more data is expected.

Then when TS sends a 'g' command, it will expect in response only 6 bytes (wrapped in the envelope f coarse). The bytes are expected in the order they were defined, but TS always writes them in offset order.

Some things to note on the scatteredOffsetArray:

- This is normally set on a separate RAM only page with no burn command. It is expected that the scatteredOffsetArray is uninitialized until TS writes something to it.

- TS will update this array if subscriptions change, the size of the returned data doesn't match what TS expects or if a protocol error of 0x93 is received.

0x93 indicates the scatteredOffsetArray is not set or contains no data.

12 [GaugeConfigurations]

The gauge configuration section is used to define Gauge Categories and the gauge templates available for use in other sections and on the right click menu of all gauge clusters. The rendering of a gauge is independent of the defined template and up to the application or gauge style.

12.1 Gauge Categories

Like all ini sections, the GaugeConfigurations section is read in order line by line. You can assign a group of gauges a category so the UI can then use this for grouping template. If there is no category assigned, the application will generate generic template category names.

To set the category name for a set of gauge templates, use the tag:

```
gaugeCategory = "Some Category Name"
```

Once a line sets a category name, all gauge templates defined on subsequent rows will be assigned to that category until a new category is set.

12.2 Gauge Templates

Name = Case-sensitive, user-defined name for this gauge configuration.

Channel = Case-sensitive variable name defined in the OutputChannels section that engine data will be returned to display on the gauge.

Title – String or String Function for Title displayed by the gauge.
Units – String or String Function for Units displayed below value on gauge.
Lo – Numeric value or expression defining the lower scale limit of gauge.
Hi – Numeric value or expression defining the upper scale limit of gauge.
LoD – Numeric value or expression defining the lower limit at which danger color is used.
LoW – Numeric value or expression defining the lower limit at which warning color is used.
HiW – Numeric value or expression defining the upper limit at which warning color is used.
HiD – Numeric value or expression defining the upper limit at which danger color is used.
vd – Numeric value or expression defining the decimal places in displayed value
ld – Numeric value or expression defining the label decimal places for display of Lo and Hi, above.
Active – Optional expression that determines if this gauge template should currently be available. If not defined, always true

12.2.1 Example Gauge Entry

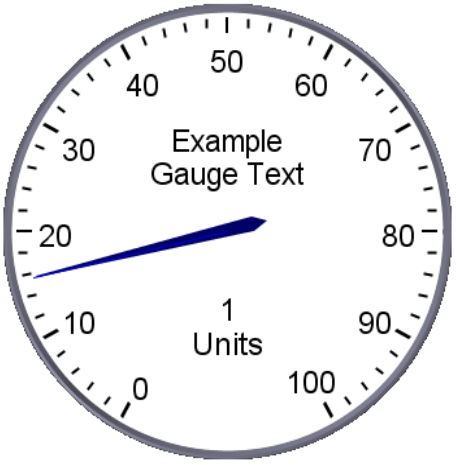


Figure 12-1

```
    Name      Channel          Title          Units   Lo   Hi  LoD  LoW   HiW   HiD  vd  ld
SampleGauge = throttle, "Example Gauge Text", "Units", 0,  100,  -1,  -1,  100,  100,  0,  0, { 1 }
```

12.2.2 Example set of gauge templates with categories

```
gaugeCategory = "Sensor inputs1"
  mapGauge      = map, "Engine MAP", "kPa", 0, {loadhigh},  0,    20,  200,  {loadhigh}, 1, 0
  fuelloadGauge = fuelload, "Fuel Load",{ bitStringValue( algorithmUnits , algorithm  ) },  0, {loadhigh},
                  0,    20,  200,  {loadhigh}, 1, 0
  fuelload2Gauge = fuelload2, "Secondary Fuel Load", { bitStringValue( algorithmUnits , algorithm2  ) }, 0,
                  {loadhigh},      0,    20,  200,  {loadhigh}, 1, 0

gaugeCategory = "Outputs"
       nitrous1_duty = nitrous1_duty, "Nitrous 1 Duty", "%", 0, 100, 100, 100, 100, 100, 0 ,0
       nitrous2_duty = nitrous2_duty, "Nitrous 2 Duty", "%", 0, 100, 100, 100, 100, 100, 0 ,0

; Warning, the above entries with expressions are wrapping, in real use it must be defined on a single line.
```

13 [DATALOG]

The DataLog section is used to define the channels available for data logging as well as formatting of those channels.

13.1.1 Entry attributes

The entries are saved in the datalog file in the order in which they appear in the list below.
Each entry in the DataLog section will begin with entry =; followed by the entry attributes.

entry = Channel , Label, Type, Format, Enabled Cond, Lag

Channel - Case sensitive name of output channel to be logged.
Label - String written to header line of log.
Type - normally float or int, no longer used.
Format - C-style output format of data or tag
Boolean tags: yesNo, onOff, highLow, activeInactive
For Hex output: hex
Enabled Cond - This field will only be logged if the enable condition is blank or resolves to true.
Lag - If for any reason you need to have a field lag n records behind realtime. Use a number or expression

When using a Boolean format, a String will be logged based on the channel value. If channel value = 0 it is false the logged String will be ("No", "Off", "Low", "Inactive") any non-zero value is considered true and ("Yes", " On", "High", "Active") will be logged depending on the flag used.

Units are logged for each entry if they have been assigned to the underlying OutputChannel
Entries can optionally be assigned to a category by adding the keyword:
category and assigning a String value. All entries after the set category will be assigned the String.
Category will only be included in the log if a log format supports it such as mlg 2.0

13.1.2 Example:

```
[Datalog]
   ;        Channel          Label          Type    Format  Enabled Cond  Lag
   ;        -----------     --------        -----   ------  ------------  ---
category = "Common"
   entry = time,            "Time",         float,  "%.3f"
   entry = seconds,         "SecL",         int,    "%d"
category = "Engine"
   entry = rpm,             "RPM",          int,    "%d"
   entry = map,             "MAP",          float,  "%.1f"
   entry = boostpsig,       "Boost psi",    float,  "%.1f"
   entry = throttle,        "TPS",          float,  "%.1f"
   entry = maf,             "MAF",          float,  "%.01f", { MAFOption }
   entry = mafload,         "MAFload",      float,  "%.1f",  { MAFOption }
   entry = table3Active,    "VE Table 3",     int,  "activeInactive"
   entry = pinH1,         "Pin H1 state",     int,  "highLow"
```

In actual use you may commonly have far more.

14 [MENU]

Menu entries will be displayed as actionable menus to the user. The [Menu] section allows you to define menu entries. Menus can be placed on main application tool bar, any defined dialog tool bar as well as the main window calibration toolbar. The Calibration Tool bar is the primary space for calibration related menu items.

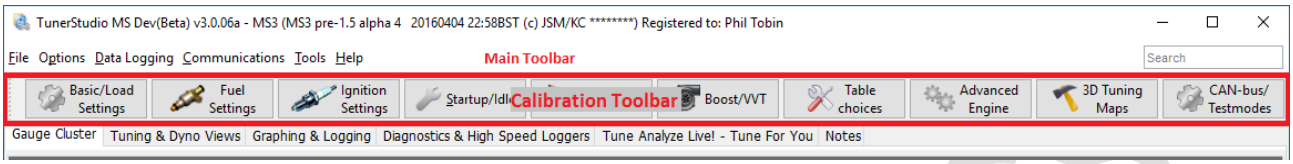


Figure 14-1 Toolbar Navigation

Each menu item can reference and open in a dialog any of the following:

- Any dialog defined in [UserDefined] or [UiDialogs]
- Curve Editor defined in [CurveEditor]
- 2D Table Defined in [TableEditor]
- 3D Table tuning dialog defined in [TableEditor]
- Standard Dialogs that are specific to the ECU being serviced

14.1 STANDARD DIALOGS

Standard dialogs are dialogs that are part of the application. The design and layout is built into the application, but they are launched as needed via menu items that reference the standard dialog name. Some of these dialogs have no ECU Definition configuration and are not flexible enough to be used other than for a specific firmware family, others can be configured to work with many firmwares.

14.1.1 Standard Dialogs

- std_injection
- std_realtime
- std_accel
- std_ms3Rtc
- std_ms3SdConsole
- std_ms2gentherm – Configurable, see [ReferenceTables] section
- std_ms2geno2 – Configurable, see [ReferenceTables] section
- std_constants
- std_warmup
- std_port_edit – Configurable, see [PortEditor] section
- std_trigwiz

14.1.2 Additional std_ keyword

- std_separator – this allows you to place a separator on the menu for grouping of menu items.

14.2 Defining Menus

There are 3 primary keywords for defining Menus:

- menuDialog – set the window or view
- menu – set or create the menu to append subMenu items
- subMenu – define a new actionable menu to be added to the set menu
- groupMenu – define a Menu to be listed under the set menu and hold child menus
- groupChildMenu – same as subMenu, but added to the set groupMenu

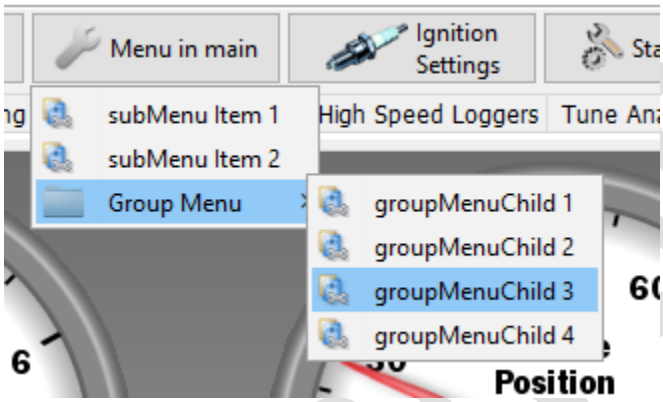


Figure 14-2

14.2.1 menuDialog

Used to set the window or view. Once a menuDialog is set, it will be used for all subsequent menu definitions until set to a different window or view. The value can be the name of any dialog defined in [UiDialogs] /[userDefined] or the keyword main if the menu is to show on the main application window.

14.2.1.1 Usage:

```
menuDialog = main
;or
menuDialog = definedDialogName
```

14.2.2 menu

The menu keyword is used to set the top level menu name if a menu by the set name already exist on the set menuDialog, that menu will be used for subsequent subMenu entries, otherwise a new menu will be created and added to the set menuDialog.

On the menuDialog main, the application has predefined main menu bar menus, by setting the menu to one of the predefined application menus, your menu will always be added to that menu. However, when set to any other names, where that menu is created will be determined by the user selected navigation. The default navigation “Toolbar Style” will create a new button as seen in Figure 14-1 Toolbar Navigation.

On the main window, the following menus will already on the window:

- File
- Options
- Data Logging

- Communications
- Tools
- Help

If any of those are set as the menu, all subMenu items will be added to the existing menu.

14.2.2.1 Usage:

To create a Menu with the Label “Nitrous Oxide”, all subMenu and groupMenu entries to follow will be appended to this menu until a new menu is set.

```
menu = “Nitrous Oxide”
```

14.2.3 subMenu

The subMenu keyword defines the actionable menu item. The menu item will be produced with the provided label, upon user activation the targetDialog will be launched.

14.2.3.1 Usage:

The menuDialog and menu must be set prior to defining a subMenu.

```
subMenu = targetDialog,      "Menu Label"
```

14.2.4 groupMenu

A groupMenu is a Menu or folder for children among subMenu object. The groupMenu allows adding a group of childGroupMenu items.

14.2.4.1 Usage:

```
groupMenu = "Group Menu Label"
```

14.2.5 groupChildMenu

A groupMenuChild behaves the same as a subMenu, but will be appended to the last defined groupMenu as opposed to last menu set.

14.2.5.1 Usage:

```
groupChildMenu= targetDialog , "groupMenuChild Label"
```

14.2.6 Hot Keys

A hot key for a menu can be set by including an & (ampersand) within the text string label of the menu. When a hot key is set, then when a user pressed Alt+[HotKey] the menu will activate. This will be set only when the menu created, if the menu component is being set, but has already created, it will do nothing.

Caution must be taken as to not create conflicts with other items in the ECU Definition as well as the application hot keys.

14.2.6.1 Example set a Hot Key of x to a menu:

```
menu = “Nitrous O&xide”
“Nitrous Oxide will still be displayed to the user, but as Nitrous Oxide so the hot key is underlined.
```

14.2.7 Visibility and enablement

The menu, subMenu, groupMenu and groupChildMenu can optionally have enable and visibility expressions assigned. As with Components, the application will react dynamically at runtime. To use these, add 1 expression as a parameter for enable / disable and 2 if visible / invisible is desired.

14.2.7.1 Enable / Disable Example:

```
menu = "Nitrous Oxide",  { nitrousActive }
```

In this case the created "Nitrous Oxide" menu will be enabled or disabled based on the state of nitrousActive. This assumes there is an OutputChannel or Constant defined with the name nitrousActive and it will be 0 when Nitrous is not active.

14.2.7.2 Visible / Invisible

```
Menu = "Nitrous Oxide",  { nitrousActive },  { nitrousActive }
```

In this case the created "Nitrous Oxide" menu will be visible or invisible based on the state of nitrousActive. This assumes there is an OutputChannel or Constant defined with the name nitrousActive and it will be 0 when Nitrous is not active.

Note there still must be an enable / disable expression as a place holder even if it is a redundant or empty {} entry.

14.3 Example Menu Definition

```
menuDialog = main ; set to the main window
 menu = "Menu in main"; set the menu
     subMenu = targetDialogName,   "subMenu Item 1"
     subMenu = targetDialogName2,   "subMenu Item 2"
      groupMenu = "Group Menu"
          groupChildMenu= dialogName1 , "groupMenuChild 1"
          groupChildMenu= dialogName2 , "groupMenuChild 2"
          groupChildMenu= dialogName3 , "groupMenuChild 3"
          groupChildMenu= dialogName4 , "groupMenuChild 4"
```

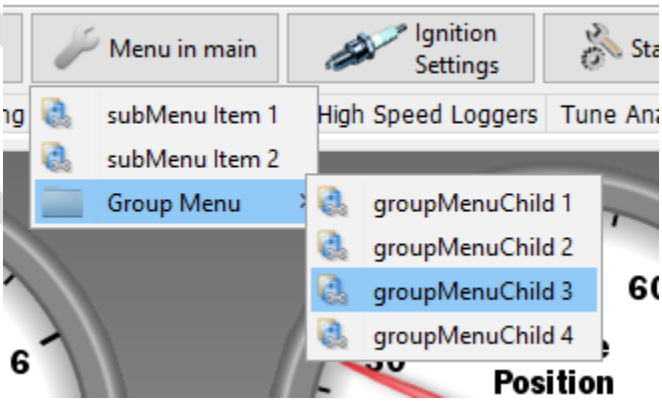


Figure 14-3

This same menu can be moved to the VE Table Dialog by simply changing the menuDialog

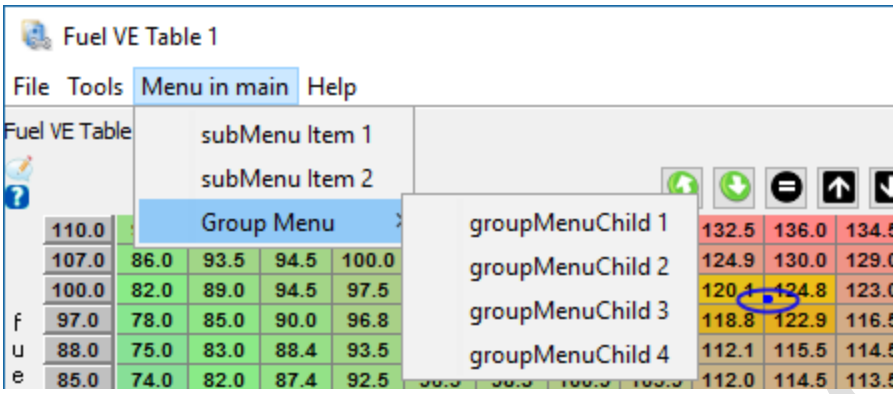


Figure 14-4

```
menuDialog = veTable1Tbl
    menu = "Menu in main"
             subMenu = targetDialogName,    "subMenu Item 1"
             subMenu = targetDialogName2,    "subMenu Item 2"
           groupMenu = "Group Menu"
             groupChildMenu= dialogName1 , "groupMenuChild 1"
             groupChildMenu= dialogName2 , "groupMenuChild 2"
             groupChildMenu= dialogName3 , "groupMenuChild 3"
             groupChildMenu= dialogName4 , "groupMenuChild 4"
```

15 [UserDefined] – [UiDialogs]

15.1 Overview

The primary purpose of this section is to define the dialogs and panels containing the settings your wish to present to the user. In the most basic form, dialogs are defined that contain references to scalar, bit and string Constants. The Application will use the Meta data of these Constants to present them as UI components within a dialog / panel. Each dialog defined can then be invoked directly from a defined [Menu] or [KeyAction] by referencing the dialog name as the target. You can also use each dialog as a panel to be placed on other dialogs using several layouts to create more complex views. In addition to dialogs defined here, you can place Curve Editor, Table Editor and Standard Dialogs on these complex views. Visibility and enablement for all dialogs, panels, sub-panels and row items can be defined using standard expressions.

15.2 Section keywords

15.2.1 Top level keywords

Top level keywords start the definition of a new entity; the new entity may have keywords of its own to further configure attributes that are defined following the initial definition

- dialog
  - topicHelp – set a help reference that will check for a local copy or retrieve from Internet
  - webHelp – launch a web browser to a set url
  - panel – Add a child dialog to the dialog being defined

- field* – render a scalar, bit or string Constant in the default manor
- gauge – Place a single round dial in the dialog.
- liveGraph – Place a multi-line graph in the dialog.
  - graphLine – define 1 or more lines on a liveGraph
- logFieldSelector* – MegaSquirt3 SD card field selector
- settingSelector* – Define a drop down of option that will set 1-n Constant values
  - settingOption – set a predefined set of constant values for a settingSelector
- radio* – Render a bit Constant using radio buttons as opposed to dropdown.
- channelSelector* – Drop down list of OutputChannels
- canDeviceSelector* – Dropdown list of ECU's set up in the project
- canClientIdSelector* – Screens MegaSquirt CAN ID's for friendly dropdown
- slider – Render a Scalar Constant as a slider as opposed to numeric input
- commandButton* – Render a button that will send controllerCommand(s)
- displayOnlyField* – Display scalar and bit Constants in a read only fashion
- array1D – Display a list of numeric inputs for a 1D array
- indicatorPanel – Display a group of indicators that will have an on/off state based on an expression
- readoutPanel – Create a group of runtime value readouts that can be placed in the dialog like a panel.
  - Readout – the definition for each readout added to a readoutPanel
- runtimeValue – Display an OutputChannel value in a dialog. Entry is the same as a field, but provide an OutputChannel name instead of a constant

** RowItem – basic component typically placed on a basic yAxis dialog.*

15.3 User Help

```
topicHelp = "file://$getProjectsDirPath()/docs/HelpDoc_x.x.pdf#xxx"
```

topicHelp – A optional entry that can be assigned to any dialog, Curve Editor or Table editor that will enable a help menu on the defined dialog for help specific to the settings in that dialog. This can be used in 3 ways:

15.3.1 URL

Set a standard url to a page or pdf on the internet:

```
topicHelp = "http://helpwebsite.com/helpdocs/help.html#anchor"
```

When used in this manor; upon user action the url will be opened in a web browser.

15.3.2 Local Cached PDF

Set a file URL to open a pdf from the local file system. The application will check the local file system for the referenced document. It is found it will be opened, if not and internet is available the application will use the defined helpManualDownloadRoot to download it from the internet to the local location for current and future access. Deep links and anchors are supported.

a helpManualDownloadRoot should be defined in the [TunerStudio] section

Example:

```
topicHelp = "file://$getProjectsDirPath()/docs/HelpDoc_x.x.pdf#xxx”
```

With the above entry, the the portion of the url “file://$getProjectsDirPath()/docs/” resolves to the local dir [ProjectsDir]/docs and pdf document will be stored in this local directory by the name HelpDoc_x.x.pdf. This allows it to be accessed by multiple projects without repeated download and will keep your help documents in a single location. If the file is not found at this local location, the application will attempt to download the pdf from the url helpManualDownloadRoot/HelpDoc_x.x.pdf and store it locally for offline use.

Once the local cached pdf file is available, it will be displayed using an integrated PDF viewer. Anchors are honored.

15.3.3 Plain Text help

Plain text help can be defined in this section, then accessed by name from a [Menu].

Example entry:

```
help = helpReferenceName, "Help Title"
   webHelp = "http://helpwebsite.com/helpdocs/help.html#anchor" ; optional webHelp
   text = "This is help text that the user can see if the select the"
   text = "menu defined in the [Menu] section referencing helpReferenceName "
```

15.4 Defining Dialogs

Once a Dialog is defined, it can then be used as a panel to be added to other dialogs, so in some cases they will be referred to interchangeably as dialogs and panels.

The definition of a dialog will always begin with a row using the keyword dialog:

```
dialog = dialogName,  “Dialog Title”, dialogLayout, password
```

dialogName – required and used to reference this dialog elsewhere
“Dialog title” – Optional. If present a border with title will be displayed.

- A title of “.” Will create a border with no title.
- An empty string “” title will remove the border and title.

dialogLayout – Optional, set a layout as defined in the Dialog Layouts section. By default yAxis will be used.
password – Set a password that will be required to open the dialog. Generally for use with encrypted ECU Definition files.

All subsequent lines are assumed to be defining a component that belongs to this dialog until another row starting with the keyword “dialog” is found. A dialog can contain 0-n components with the limit based on screen space or the maximum supported by the dialog layout used.

15.4.1 Dialog Components

Dialog Components are any item you are able to add to a defined dialog. They generally fall into 2 categories:

1. Row Item Component – Are added directly to a dialog definition as the next component within the dialog layout.

2. Panel Component – The panel keyword allows you to embed additional larger components into a dialog. This can include other dialogs defined in this section, Table Editors, Curve Editors and Standard Dialogs.

Row Item Components allow the addition of Constants to dialogs with a simple one line entry. The addition of the Panel Component provides a means to add other components as well as nest dialogs within each other. By gaining a good understanding of the dialog layouts, there are few limits of how you can present a group of settings to the user.

15.4.2 Adding Components to a dialog

15.4.2.1 field

field is the most basic and commonly used component. A field will have a label and reference a single bit, scalar or string Constant and render it in the default manor based on the Constant attributes.

- bit – Rendered as a dropdown with the provided list of selections as defined in the bit Constant section of this document.
- Scalar - Rendered as a numeric text input applying the limits and display format defined in the Constants section.
- String – Rendered as a free form text entry and enforcing length

Format of adding a field:

```
field = "User Label", someConstant, { enabledExpression }, {visibleExpression }
```

Only attribute 1 is required, the label. The label can also contain a String Function to change based on conditions.

If there is no Constant defined, the label will be displayed using the full row. If enabledExpression or visibleExpression are not defined, they are always assumed to be true.

Additional label red and blue coloring can be applied by prefixing your string with a # or !. If there are units defined for the underlying Constant, they will be appended to the label within parenthesis.

Thus this code snippet will produce the dialog below:

```
 dialog = map_sample_dialog, "MAP Sample Settings"
    field = "My Full Row Label"
    field = "!A red setting label", mapsample_opt2
    field = "#A blue setting label",     mapsample_window
    field = "A normal setting label", mapsample_angle
; Note, mapsample_opt2, mapsample_window and mapsample_angle would be Constants
```

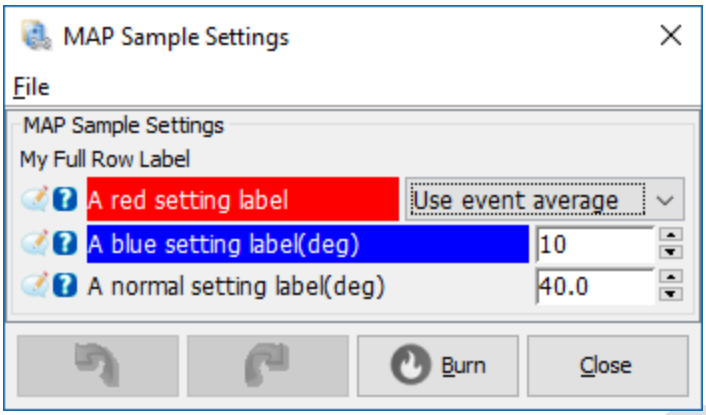


Figure 15-1

15.4.2.2 radio

The radio component provides an alternative way to render a bit Constant. Only bit type Constants are valid. The radio buttons can be laid out horizontally or vertically.

Format of a radio entry:

```
radio = orientation, "Label Text", bitConstant, { enable }, { visible }
```

Required:

Orientation – Defines the direction they are added. Must be either vertical or horizontal.
Label – Can be an empty string "" if no label is desired.
bitConstant – The name of a bit Constant defined in the [Constants] section.

Optional:
enable – The expression driving enable / disable. If not present always true.
visible – The expression driving visibility. If not present always true.

By changing the top bit Constant in the previous dialog to a radio, this is the resulting dialog:

```
dialog = map_sample_dialog, "MAP Sample Settings"
    field = "My Full Row Label"
    radio = horizontal, "A radio label", mapsample_opt2
    field = "#A blue setting label",     mapsample_window
    field = "A normal setting label", mapsample_angle
```

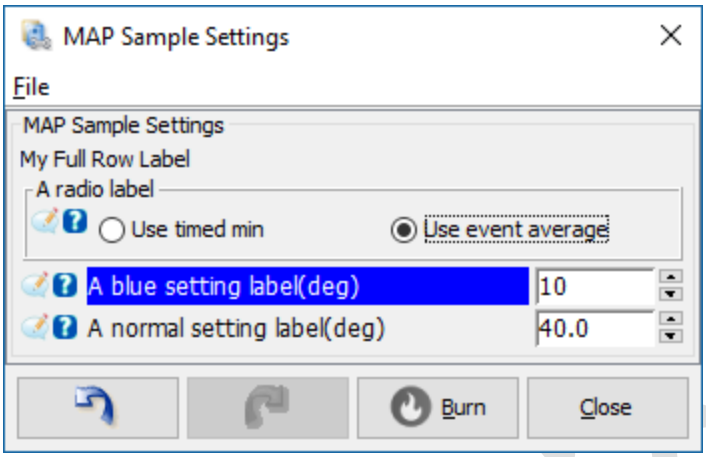


Figure 15-2

15.4.2.3 displayOnlyField

The syntax of a displayOnlyField works the same as a field. However, the value will be rendered as a read only label and will not be editable by the user. This is valid for bit or scalar Constants

```
dialog = map_sample_dialog, "MAP Sample Settings"
    field = "     My Full Row Label"
    displayOnlyField = "!A red setting label", mapsample_opt2
    field = "#A blue setting label",     mapsample_window
    field = "A normal setting label", mapsample_angle
```

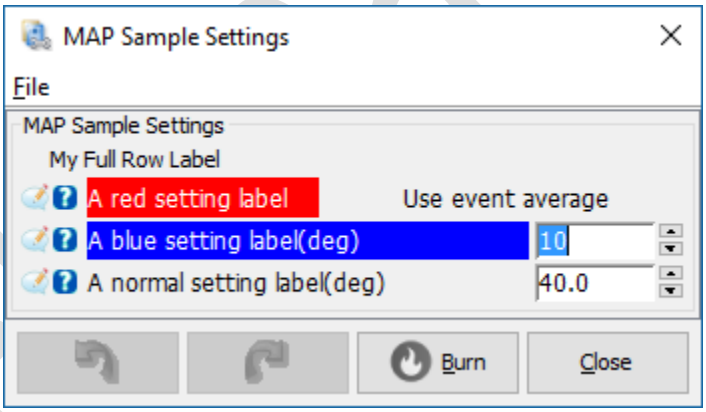


Figure 15-3

15.4.2.4 slider

The slider Component provides an alternative presentation for scalar Constants. Instead of presenting a numeric text input, a slider widget will be displayed with the value as a label. Sliders can be vertical or horizontal.

Example Syntax:

```
dialog = exampleDialog, "Example Dialog Title Text", xAxis
     slider  = "Short Description", scalarConstant, horizontal
     slider  = "Short Description", scalarConstant, vertical
```

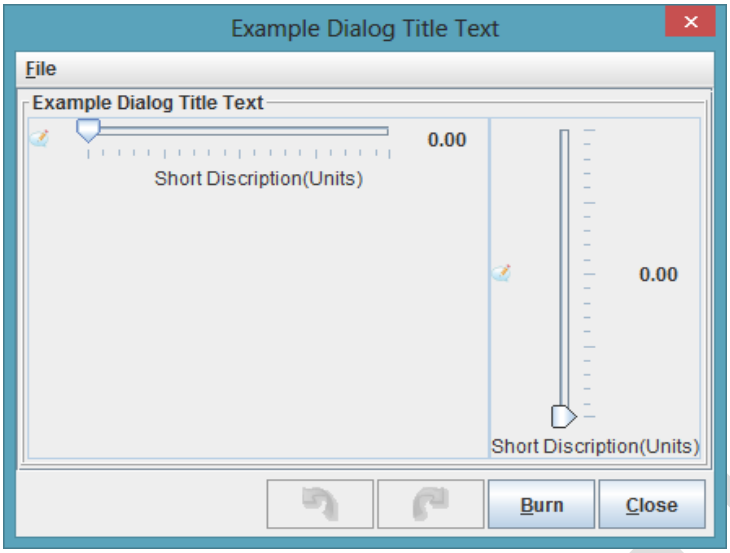


Figure 15-4: Slider Example

15.4.2.5 settingSelector

A settingSelector allows you to combine a list of preset values for a set of scalar Constants into a single drop down.

The row defining a settingSelector has only a Label as a parameter. The keyword settingOption is then used to add the values to the dropdown. You may add as many settingOptions to a settingSelector as desired.

settingOption entries are made up of a user label and 1-n name value pairs. Upon selecting an Option from within the UI, the name / value pairs will be used to set the Constant for all defined for that settingOption.

Upon initializing the UI will look through all settingsOptions for one that matches the current settings and set the drop down to the matching option. If no match is found, it will be set to a display of Custom allowing the user to enter their own values.

Sample dialog with a settingSelector:

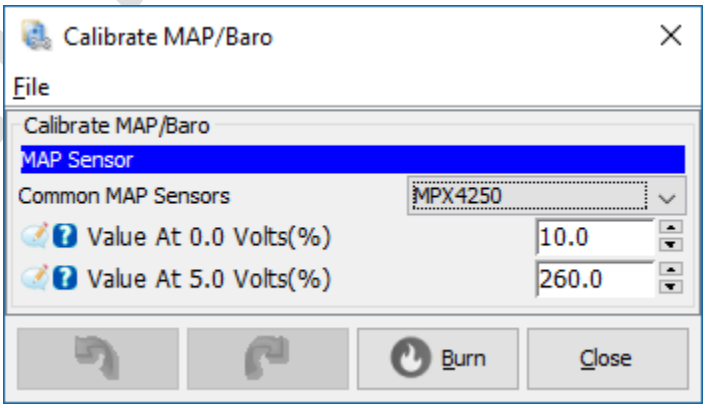


Figure 15-5

Dialog Definition:

```
dialog = sensorCal, "Calibrate MAP/Baro"
   field = "#MAP Sensor"
   settingSelector = "Common MAP Sensors"
      settingOption = "MPX4115",  map0=10.6, mapmax=121.7
      settingOption = "MPX4250",  map0=10,   mapmax=260
      settingOption = "GM 1-BAR", map0=10,   mapmax=105
      settingOption = "GM 2-BAR", map0=8.8,  mapmax=208
      settingOption = "GM 3-BAR / MPXH6300", map0=1.1,  mapmax=315.5
      settingOption = "MPXH6400", map0=3.5,  mapmax=416.5
      settingOption = "AEM 3.5 BAR", map0=-42.3,  mapmax=386.3
      settingOption = "AEM 5.0 BAR", map0=-64.6,  mapmax=581.7
   field =    "Value At 0.0 Volts",   map0
   field =    "Value At 5.0 Volts",   mapmax
```

The Constants map0 and mapmax are scalar Constants defined in the [Constants] section.

15.4.2.6 gauge

The gauge component allows you to place a single round dial on a dialog. You can place multiple gauge components on a single dialog.

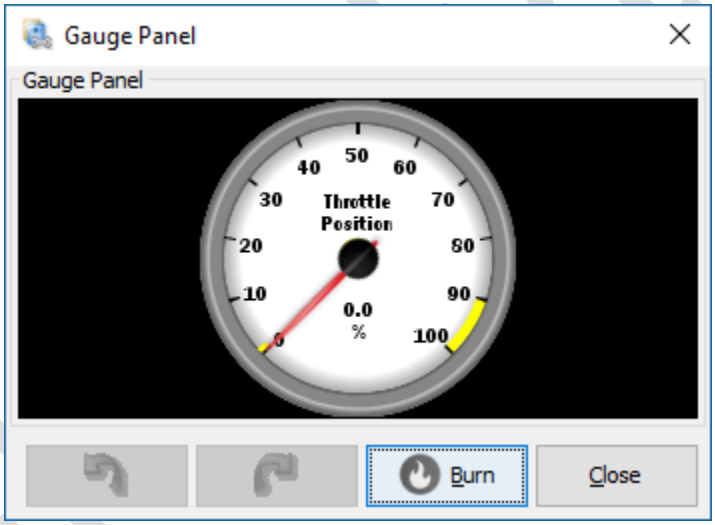


Figure 15-6

Syntax:

```
dialog = exampleDialog_gauge_panel, "Gauge Panel"
     gauge = throttleGauge
```

throttleGauge is defined in [GaugeConfigurations]

15.4.2.7 array1D

The array1D component is really more of a macro that a single component. Its usage is to generate a group of numeric inputs for a 1D array Constant. You can provide a Label, that can include the element number using the %INDEX% keyword. The %INDEX% keyword will be replaced at runtime with the array index+1 to keep counting as most end users would expect it. In most cases a Curve Editor is used and is a preferred presentation.

Syntax:

```
dialog = frpmTableBins, "RPM Table Bins for Fuel Tables"
   array1D   = "", "Fuel RPM  %INDEX% ", frpm_table
```

frpm_table is a 1D Array defined in the Constants section.

An unused empty string is required before label for legacy support.

This dialog will be render as below:

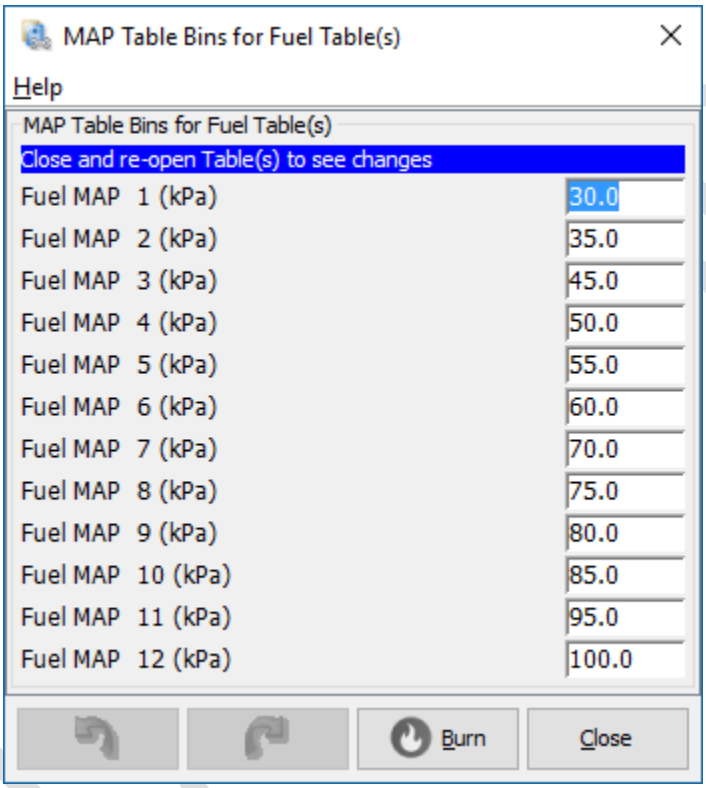


Figure 15-7

15.4.2.8 liveGraph

The liveGraph Component defines a line graph with 1 or more runtime values to be displayed on the graph. It can be placed within the dialog using various layouts, but will typically be the North or South component of a border layout. The definition row of a liveGraph contains the name, title and any placement the layout may require. It is then followed by graphLine rows.

Each graphLine row is required to have an OutputChannel defined, then optional untits, min/max and auto scale tags. If no mins and maxes are defined, the application will look to any gauge defined in the [GaugeConfigurations] section and default to those units and min / max values.

```
;liveGraph = graphName, "Graph Title", layoutPlacement
 liveGraph = timeaeGraph, "AE Graph", South
```

```
 graphLine = afr1
 graphLine = TPSdot, "%", -2000, 2000,    auto, auto
 graphLine = MAPdot, "%", -2000, 2000,    auto, auto
;graphLine = Channel,units, min, max,  autoMin, autoMax
```

In this rendering, the 1st graphLine is assigned afr1 with no other values, thus it will set units and min/max from the Gauge Template defined in [GaugeConfigurations] that is also assigned the afr1 OutputChannel.

graphLine 2&3 have set units, min, max and auto scale set. These would be used over any Gauge Templates. The min and max values are initialized to those defined.

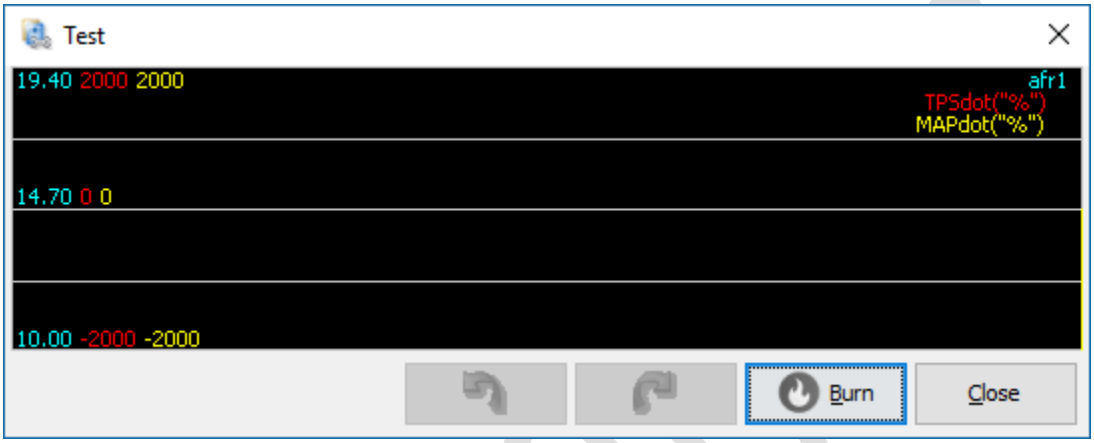


Figure 15-8

The auto scale flags allow the liveGraph to automatically adjust the min and max based on witnessed run-time data. Thus limits will change once live data is received.

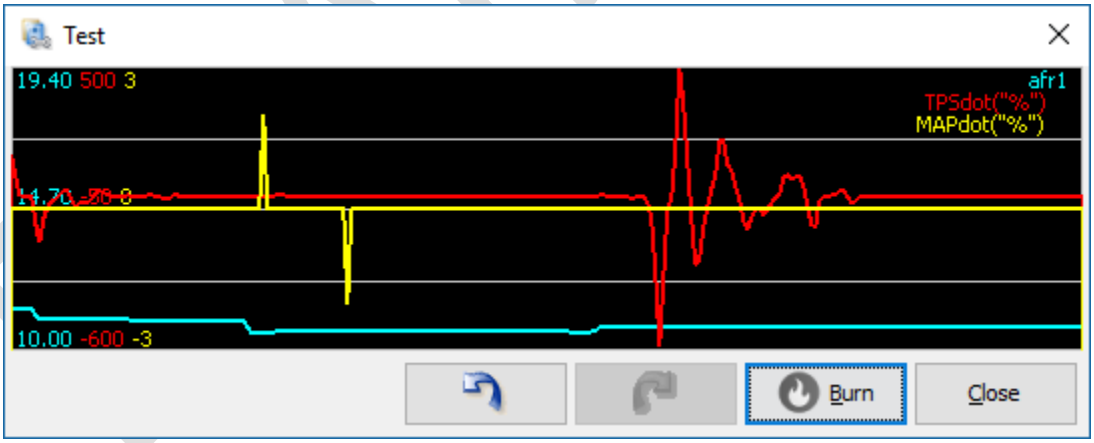


Figure 15-9

15.4.2.9 panel

The panel component allows you to place any other defined dialog, Table Editor, Curve Editor or standard dialog within a new defined dialog. An enable expression can be defined for each panel placed in a dialog that will be evaluated during any change in conditions to adjust the enabled state of all components in the

panel. By using the various layouts described in this document with multiple dialogs defined as snippets, there is a great deal of control to present settings to the user exactly as you want them.

Syntax:

```
dialog = dialogName, "", dialogLayout
    panel = complexComp1, West, {enableExpression}
    panel = complexComp2, East, {enableExpression}
```

complexComp1 and complexComp2 can be any other dialog previously defined, a Table Editor, Curve Editor, Standard Dialog.

Note: All sections are read in order, any dialog referenced by panel, must be defined before this dialog definition. The [TableEditor] and [CurveEditor] sections are loaded before [UiDialogs] therefore will always be available.

See the Dialog Layouts section of this document for examples of dialogs with panels and layouts.

15.4.2.10 commandButton

The commandButton component allows the placement of an actionable button on a dialog. It will be rendered as a Button with your defined label. When clicked, it will send a command defined in the [ControllerCommands] section of the ini file.

Syntax:

```
commandButton = "Label Text", command, { Enabled Condition }, optionalFlags
```

- command – Name reference to Controller Command defined in the [ControllerCommands] section.
- Enable Condition – Optional; any expression to determine if the Button should be enabled or not.
- optionalFlags – Append a comma delimited list of non-conflicting options.
    - clickOnClose – If set, this will always send the command when the dialog is closed
    - clickOnCloseIfEnabled – If set, will send the command if the enable condition is true
    - closeDialogOnClick – if set, the dialog will be closed after clicking the commandButton
    - showMessageOnClick – If set a message will be displayed to the user after clicking. This option must always be followed by an attribute containing a string message to be displayed.

Example:

```
commandButton = "Reboot",  cmdReboot, showMessageOnClick, "Controller Restarted"
```

cmdReboot is defined in the [ControllerCommands] section with an instruction that will reboot the controller.

This button upon clicking will send the instruction to the controller, then popup a message "Controller Restarted"

15.4.2.11 logFieldSelector

The logFieldSelector component is a complex component to relate DataLogFields with OutputChannels when using the Basic request/reply protocol. By providing 2 equal length array Constants, it will display all DataLogField entries to the user as available and allow them to select a subset. The 2 provided arrays will be populated with the offsets and lengths defined for all OutputChannels required to produce the selected

DataLogFields. When data log fields that are based on formula based OutputChannels, this component will find all under lying Controller OutputChannels required to produce the field and add them to the arrays.

It will allow the user to add DataLogFields until one of the following occur:

- The offsetConstant len is not sufficient
- The total of all values in the lenConstant is equal to maxBytes expression value

Note: As there is no record of selected DataLogFields, but actually all the underlying Controller OutputChannels required to produce the DataLogField, the selected fields may not be exactly what the user selected. All other DataLogFields that have all channels needed will appear in the "Selected Datalog Field" list whether the user specifically selected it or not.

For example:

AFR1 Error is based on a formula OutputChannel requiring AFR1 and AFR1Target.

```
afr1Error = { afr1 - afr1Target }
```

So if a user adds AFR1 and AFR1 Target to the Selected Fields, AFR1 Error will implicitly be added also.

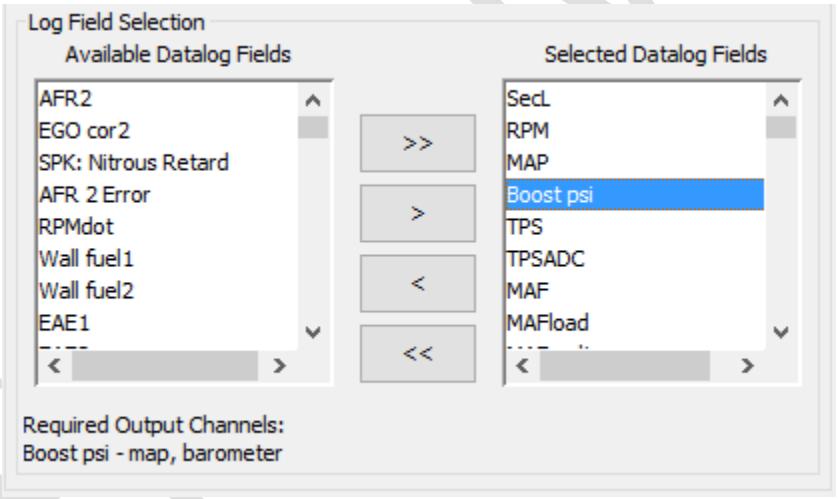


Figure 15-10

Syntax:

```
logFieldSelector = selectorName, "Title", offsetConstant, lenConstant, {maxBytes}
```

- selectorName – The name of the defined Log Field Selector to be used when referencing or adding this to a dialog.
- "Title" – The Title to be displayed at the top, "Log Field Selector"
- offsetConstant – A 1D array defined in the Constants [Section], this must be large enough to fit the maximum number of channels expected to be logged.
- lenConstant – A 1D array the same length as offsetConstant. This will be populated with the size of each OutputChannel in bytes indexed to match offsetConstant.

- {maxBytes} – If there is a record limit size, this expression will limit the sum of lenConstant to this maximum.

15.4.2.12 canDeviceSelector

The canDeviceSelector Component is specific to the standard MegaSquirt MS2 and MS3 protocol. This component will produce a dropdown with all controllers configured within the application project. The assigned Constant value will be set to the CAN ID assigned to that controller in the application.

Syntax:

```
dialog = channelSelector, "Select Channel"
   canDeviceSelector = "Location of load data", device_canid
; device_canid is a defined Constant
```

15.4.2.13 channelSelector

The channelSelector Component is specific to the standard MegaSquirt MS2 and MS3 protocol. This Component will produce a Dropdown with all defined Controller OutputChannels. This Component requires 2 Constants to operate offsetConstant and sizeConstant with a third optional Constant can_idConstant

- **offsetConstant** – The offset set of the selected OutputChannel will be set in this Constant.
- **sizeConstant** – The data size in bytes of the selected OutputChannel will be set in this Constant.
- **can_idConstant** – If defined, a Constant that provides the CAN ID of the Project Controller to list OutputChannels for. If this is not defined, the current controller channels will always be displayed. If the value of the can_idChannel changes, the dropdown will automatically be repopulated.

Enable and visible expressions are honored as always.

Example Syntax:

```
dialog = channelSelector, "Select Channel"
   canDeviceSelector = "Location of load data", device_canid
   channelSelector= "Load axis", loadOffset, loadSize, device_canid
```

This example makes use of a canDeviceSelector to set device_canid and drive what controller OutputChannels are listed in the channelSelector

15.4.2.14 canClientIdSelector

The canClientIdSelector Component is specific to the standard MegaSquirt MS2 and MS3 protocol. This component will scan all CAN ID's using the MegaSquirt pass through protocol. A dropdown will be presented with a description of devices found at each CAN ID.

One Constant is required, this will contain the value of the selected CAN ID.

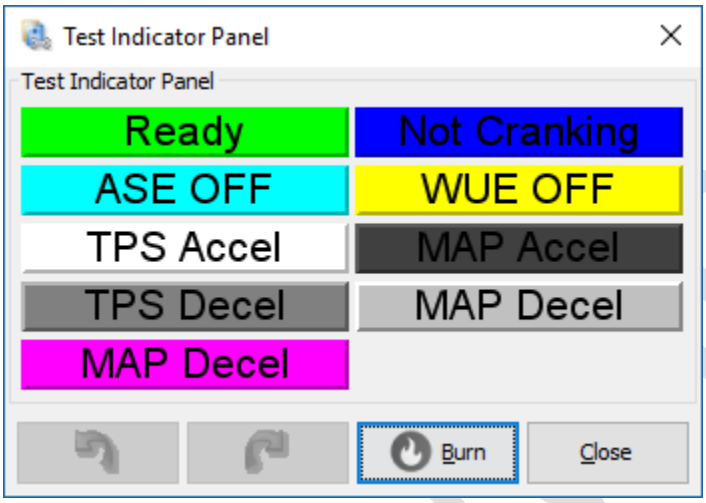
Example Syntax:

```
canClientIdSelector = "Remote CAN ID", remote_can_id
```

remote_can_id is a Constant.

15.4.2.15 indicatorPanel

An indicator panel is used to display 1 or more indicators similar to those defined in [FrontPage] for dashboards, but within the context of a dialog.



Syntax:

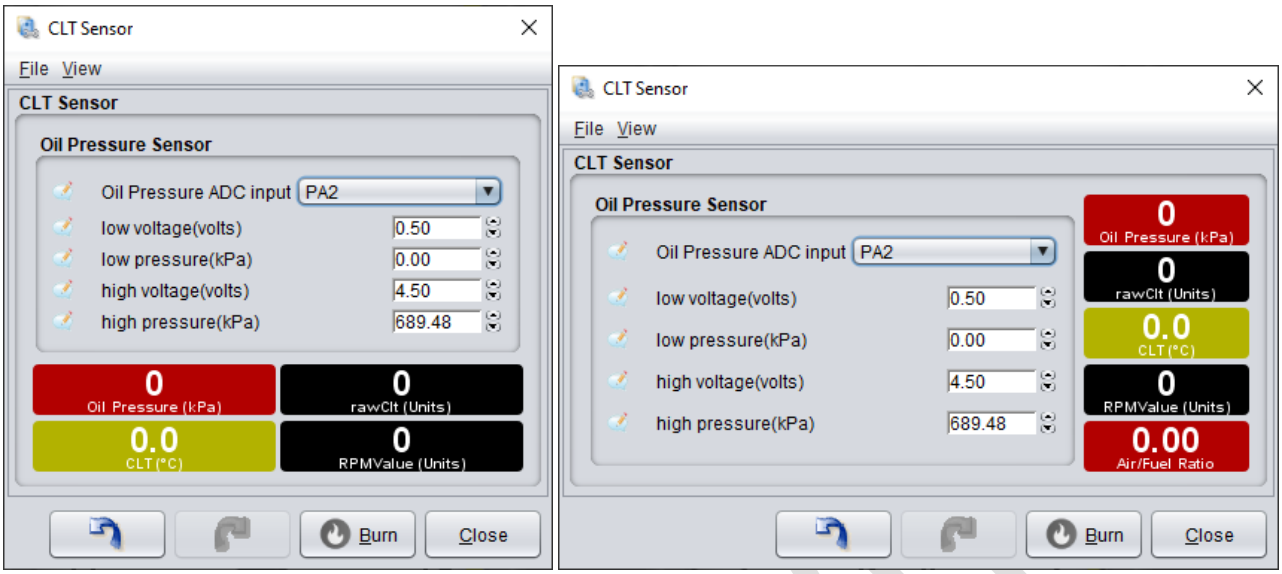
indicatorPanel = panelName, numberOfColumns, { optional enabledExpression }

This entry is then followed by definitions for the indicators to be added to the indicatorPanel. The indicator definition is the same syntax as used in the [FrontPage] section. All indicators defined in the [UserDefined] / [UiDialogs] section will be added to the last defined indicatorPanel.

Example Syntax for above dialog:

```
indicatorPanel = testIndicatorPanel, 2, { 1 }
    indicator = { ready       }, "Not Ready",     "Ready",     red, black, green, black
    indicator = { crank       }, "Not Cranking", "Cranking", blue, black, green, black
    indicator = { startw      }, "ASE OFF",       "ASE",       cyan, black, green, black
    indicator = { warmup      }, "WUE OFF",       "WUE",       yellow, black, green, black
    indicator = { tpsaccaen   }, "TPS Accel",    "TPS Accel", white, black, green, black
    indicator = { mapaccaen   }, "MAP Accel",    "MAP Accel", darkGray, black, green, black
    indicator = { tpsaccden   }, "TPS Decel", "TPS Decel",    gray, black, green, black
    indicator = { mapaccden   }, "MAP Decel", "MAP Decel",    lightGray, black, green, black
    indicator = { mapaccden   }, "MAP Decel", "MAP Decel",    magenta, black, green, black

dialog = indicatorPanelDialog, "Test Indicator Panel", border
    panel = testIndicatorPanel, Center
```

15.4.3 readoutPanel



readoutPanel adds a group of runtime readouts, basic readout type gauges, that are added to your dialog like any other panel.

Syntax:

readoutPanel = aUniqueName, numberOfColumns, [Optional Enable Condition]

This entry is then followed by as many readout entries as you desire. Readout has 3 different syntaxes.

readout = aPredefinedGaugeName ; use any gauge defined in GaugeConfiguration

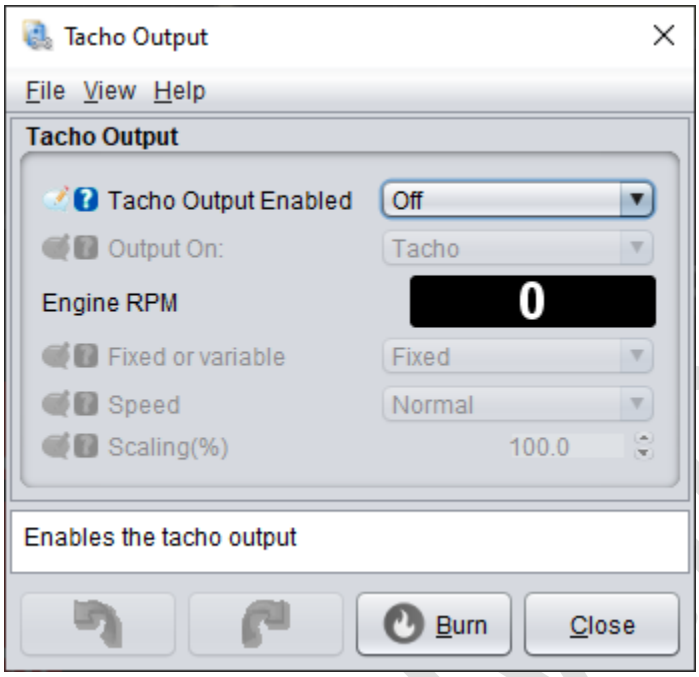
readout = anOutputChannelName ; This will have no mins/maxes.

readout = channel, Title, Units, min, max, LowDanger, LowWarn, HighWarn, HighDang, valDigits, ld
This gives the greatest control specific to this view.

Example Syntax for above readoutPanels:

```
; 2 x 2
readoutPanel = oilpReadouts2Rows, 2, { 1 }
    readout = OilPressGauge ; use the name of a predefined gauge
    readout = rawClt ; Use an OuputChannel name
            ; define the gauge here
            ; channel Title   Units  Lo     Hi  LoD  LoW   HiW   HiD    vd   ld
    readout = coolant, "CLT",   "°C", -40,  140, -15,  1,   95,   110,   1,   1
    readout = RPMValue; reference gauge defined in GaugeConfigurations

; 5 Stacked.
readoutPanel = oilpReadouts6Rows, 1, { 1 }
    readout = OilPressGauge ; use the name of a predefined gauge
    readout = rawClt ; Use an OuputChannel name
    ; define the gauge here
          ; channel  Title   Units   Lo      Hi     LoD    LoW     HiW   HiD    vd   ld
    readout = coolant, "CLT",  "°C",  -40,    140,    -15,    1,      95,   110,   1,   1
    readout = RPMValue ; reference gauge defined in GaugeConfigurations
    readout = afr1Gauge ; reference gauge defined in GaugeConfigurations
```

15.4.4 runtimeValue



Syntax:

field = “User Label”, someOutputChannel, { enabledExpression }, {visibleExpression }

While connected, this readout will continue to display the OutputChannel value.
For greater control, see readoutPanel and readout.

Example Syntax for above dialog:

```
dialog = tacho, "Tacho Output"
   field = "Tacho Output Enabled",         tacho_opt80
   field = "Output On:",  tacho_opt3f, { tacho_opt80 }
   runtimeValue = "Engine RPM" , rpm
   field = "Fixed or variable", tacho_optvar, { tacho_opt80 }
   field = "Speed", tacho_opt40, { tacho_opt80 && (tacho_optvar == 0) }
   field = "Scaling", tacho_scale, { tacho_opt80 && (tacho_optvar == 1) }
```

15.4.5 Dialog Layouts

Dialog layouts control the placement of components within the dialog. There are several supported layouts available for use when defining a dialog.

- yAxis – Place items from top of dialog to the bottom in a single column
- xAxis – place items from left to right on a single row
- border – placement of items to pull to 1 edge (North, South, East, West) or fill the Center region
- card – Stack multiple panels on top of each other with expressions to determine which is visible
- indexCard - Place items from top of dialog to the bottom in a single column, only the panel in focus is shown at full size, the remainder will be show at a fraction of full height.

15.4.5.1 yAxis Layout

yAxis Layout is the default layout and will be used in any case where there is no layout defined in the dialog definition row. This is the most common layout for adding basic row Item components that can be placed on a single row and do not support resizing.

Simple yAxis dialog layout

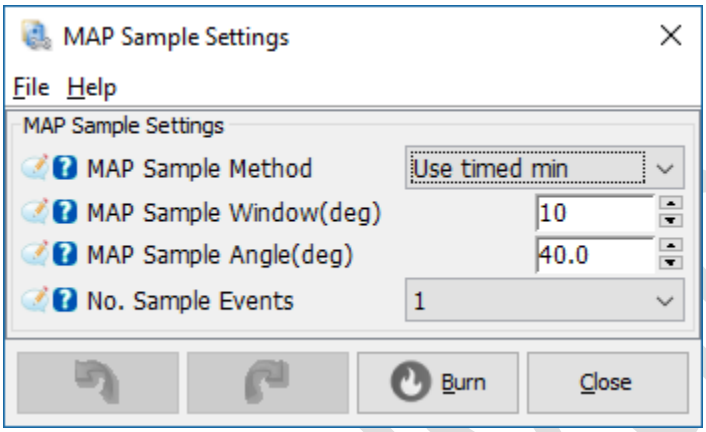


Figure 15-11

In this simple dialog there were 4 items added using an xAxis layout. They are simply placed from top to bottom with each component on a row.

15.4.5.2 xAxis Layout

xAxis Layout will place components in the dialog from left to right in a single row. The xAxis layout is most commonly used to place panels side by side.

xAxis layout placing 3 dialogs as panels in 1 dialog

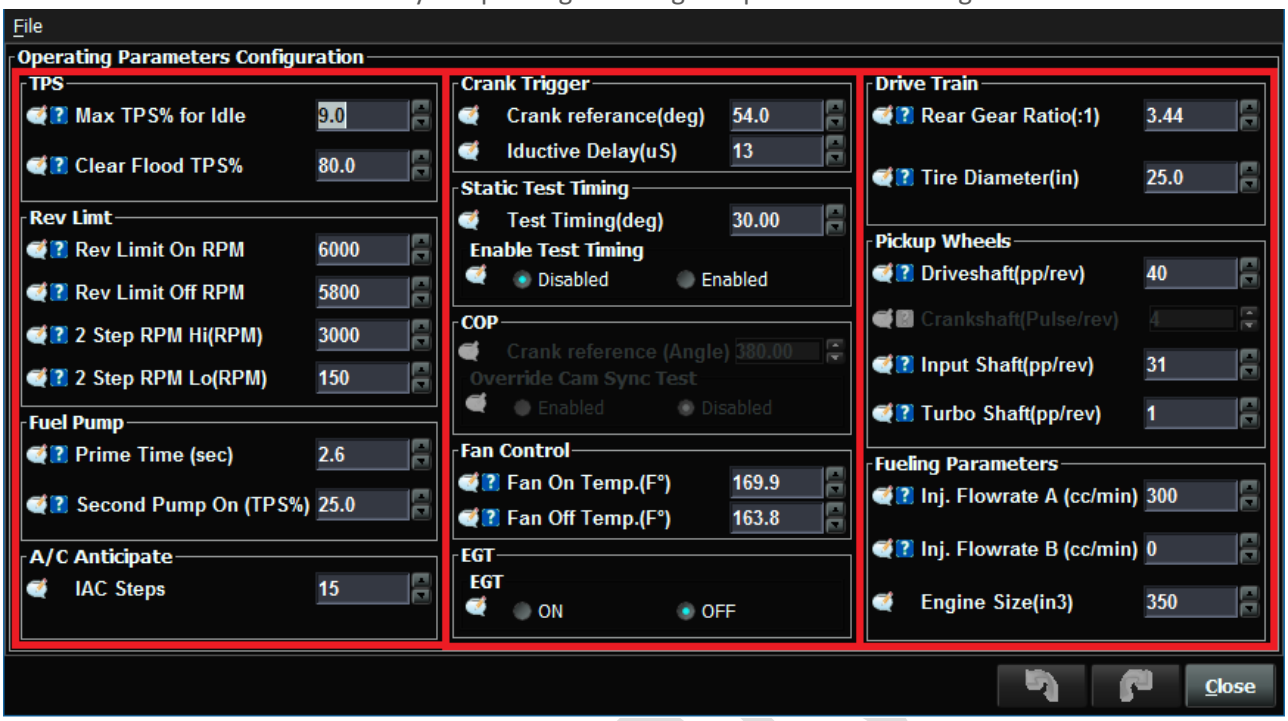


Figure 15-12

Using the xAxis layout we have placed 3 defined dialogs into a single dialog as 3 columns with equal sizing.

The xAxis layout supports 1-n components, typically other dialogs added as a panel.

Code snippet:

```
dialog = operatingPara_config, "Operating Parameters Configuration" xAxis
      panel = operatingPara_column_1
      panel = operatingPara_column_2
      panel = operatingPara_column_3
```

15.4.5.3 border Layout

The border layout provides the greatest flexibility. Not intended for use with Row Item components, but larger components such as other dialogs, Table Editors, Curve Editors, Live Graph, etc..

Up to 5 components can be added to a dialog with a border layout. Each must be added with a placement keyword of North, South, East, West or Center. Each keyword can only be assigned to 1 component per dialog and not all the keywords need be used. It is perfectly fine to only place 2 components on a dialog using a border layout.

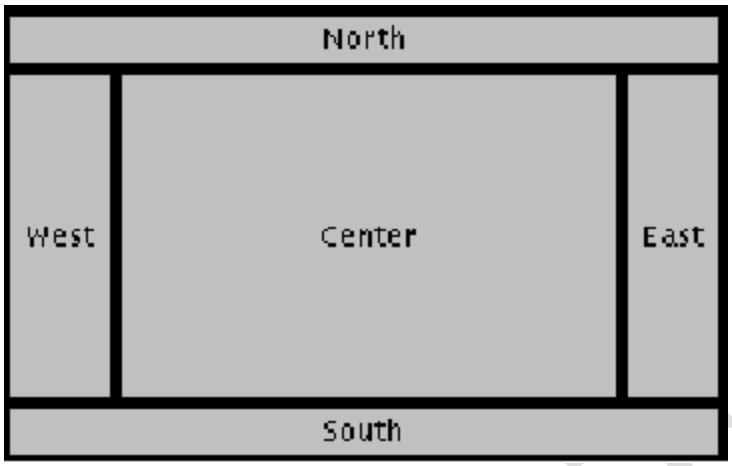


Figure 15-13

North – will place the component against the top from edge to edge giving it only the minimum space required by the component.

South – will place the component against the bottom from edge to edge giving it only the minimum space required by the component.

West – will place the component against the left side from the North component to the South component if there are North and South components, otherwise from top to bottom.

East – will place the component against the right side from the North component to the South component if there are North and South components, otherwise from top to bottom.

Center – will fill the component to all space available. This is generally the best place to put components such as Table Editors or Curve Editors as the will stretch to any dimension and nicely fill any additional space if a user should resize the dialog.

Simple border layout Dialog

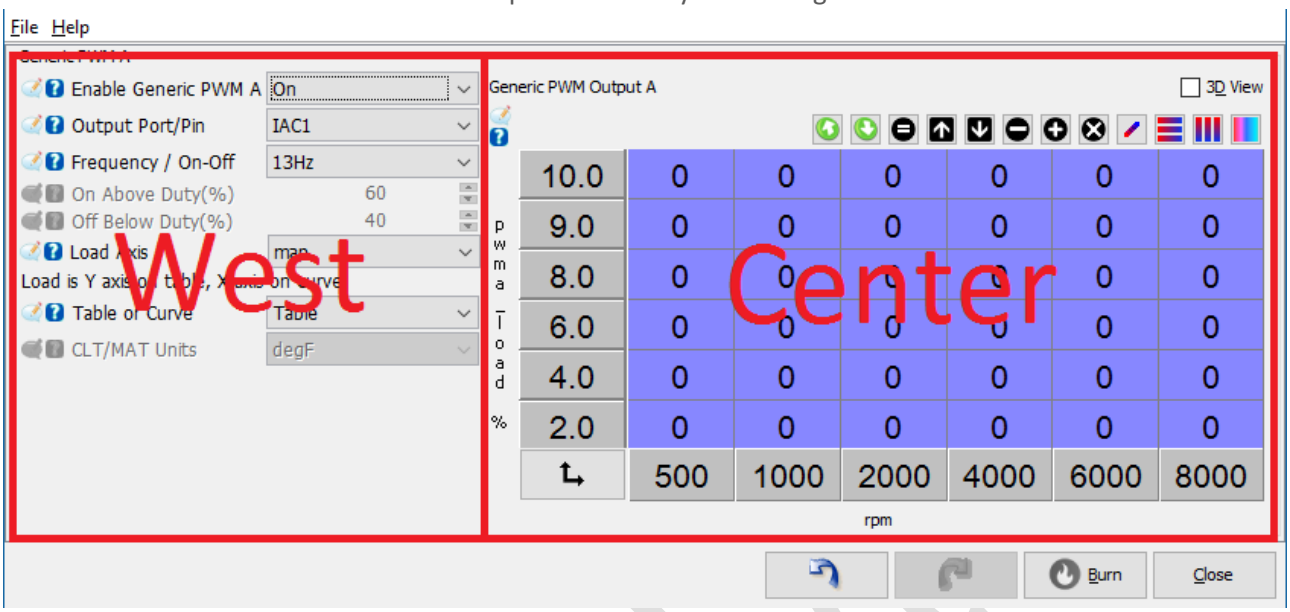


Figure 15-14

Code snippet:

```
dialog = gen_pwm_a, "Generic PWM A", border
  topicHelp = "file://$getProjectsDirPath()/docs/MS3_Reference-1.4.pdf#genpwm"
  panel = gen_pwm_left_a, West
  panel = gen_pwm_curve_grapha, Center
```

Note: gen_pwm_left_a is a dialog already defined. gen_pwm_curve_grapha is a Card Layout dialog already defined containing both a Table Editor and a Curve Editor. The Table Editor is currently visible. We will look at this example again for the Card Layout.

15.4.5.4 card Layout

Stack multiple components in the same space allowing only 1 to be visible at any time with expressions to determine which is visible.

When a dialog is defined with a card layout, 1-n components can be added as a panel, each with an expression to determine if it is the visible panel. If there are multiple panels with expressions that evaluate to true, the 1st panel with a true expression will be the one displayed.

Card Layout with Table Editor showing

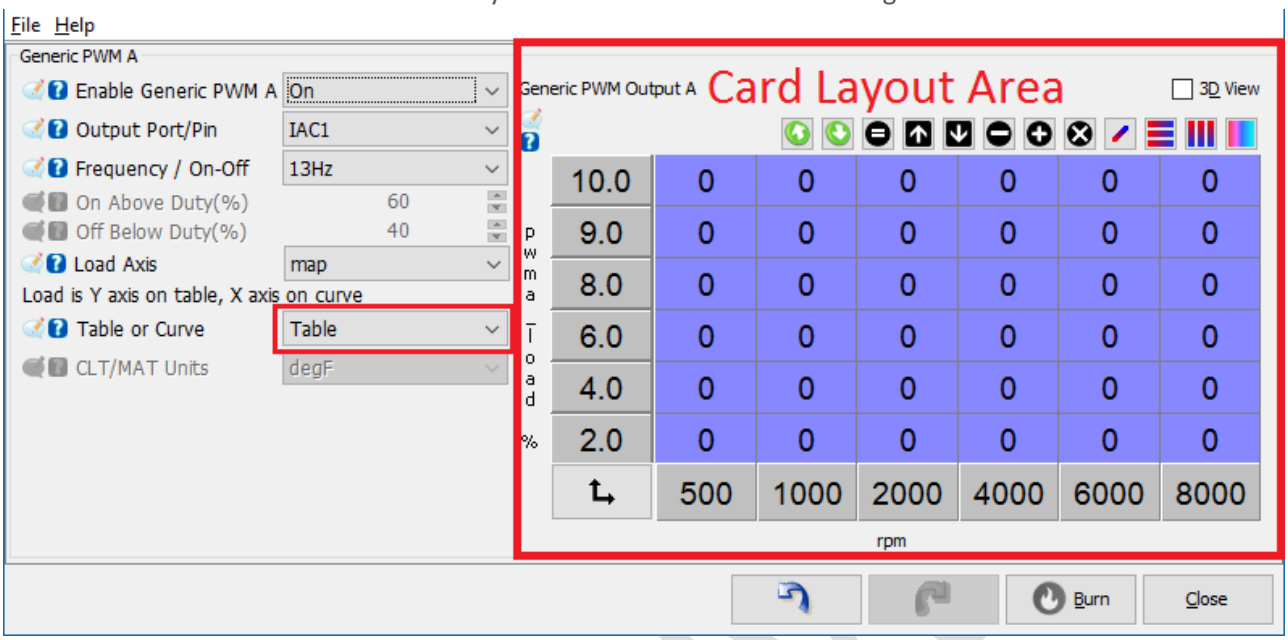


Figure 15-15

In this example, the display of the Curve Editor or Table Editor is displayed on a card layout. The table and curve are defined in the appropriate [TableEditor] and [CurveEditor] sections. The expression is based on the user selection for Table or Curve.

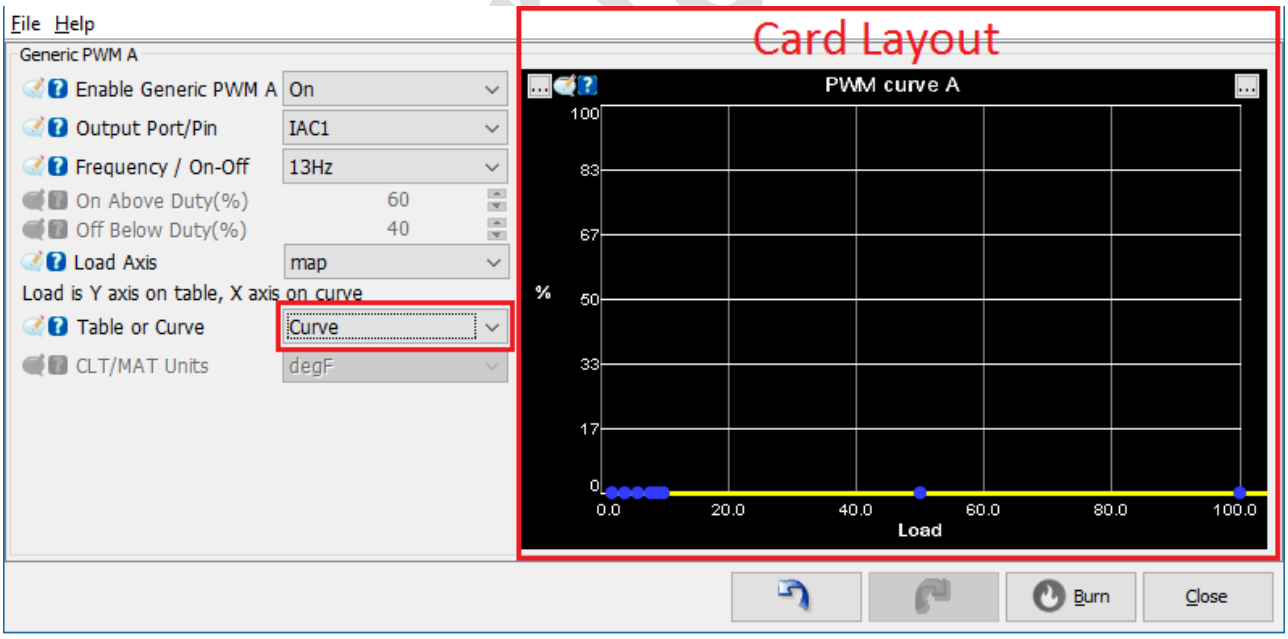


Figure 15-16

Once the user changes the “Table or Curve” selector, the underlying Constant pwm_opt_curve_a is immediately changed. The UI will detect this change and update the display at once.

Code snippet:

```
dialog = gen_pwm_curve_grapha, "", card
   panel = pwm_duties_Tbl_a, Center, {pwm_opt_on_a && (pwm_opt_curve_a == 0)}
   panel = pwm_curve_a, Center, {pwm_opt_on_a && pwm_opt_curve_a}
```

15.4.5.5 indexCard Layout

The indexCard layout is used primarily with Curve Editors but any Component can be added. This layout will place components along the Y axis from top to bottom much like the yAxis layout. However, the key difference is only the item that has focus will be displayed at full size. All out of focus items will be compressed allowing you to see them, but lessen the screen space used. Once the focus changes to another item it will become the full size component and all others will be reduced in size.

In this example a group of Curve Editors used as Shift tables are placed on a single dialog using a indexCard Layout.

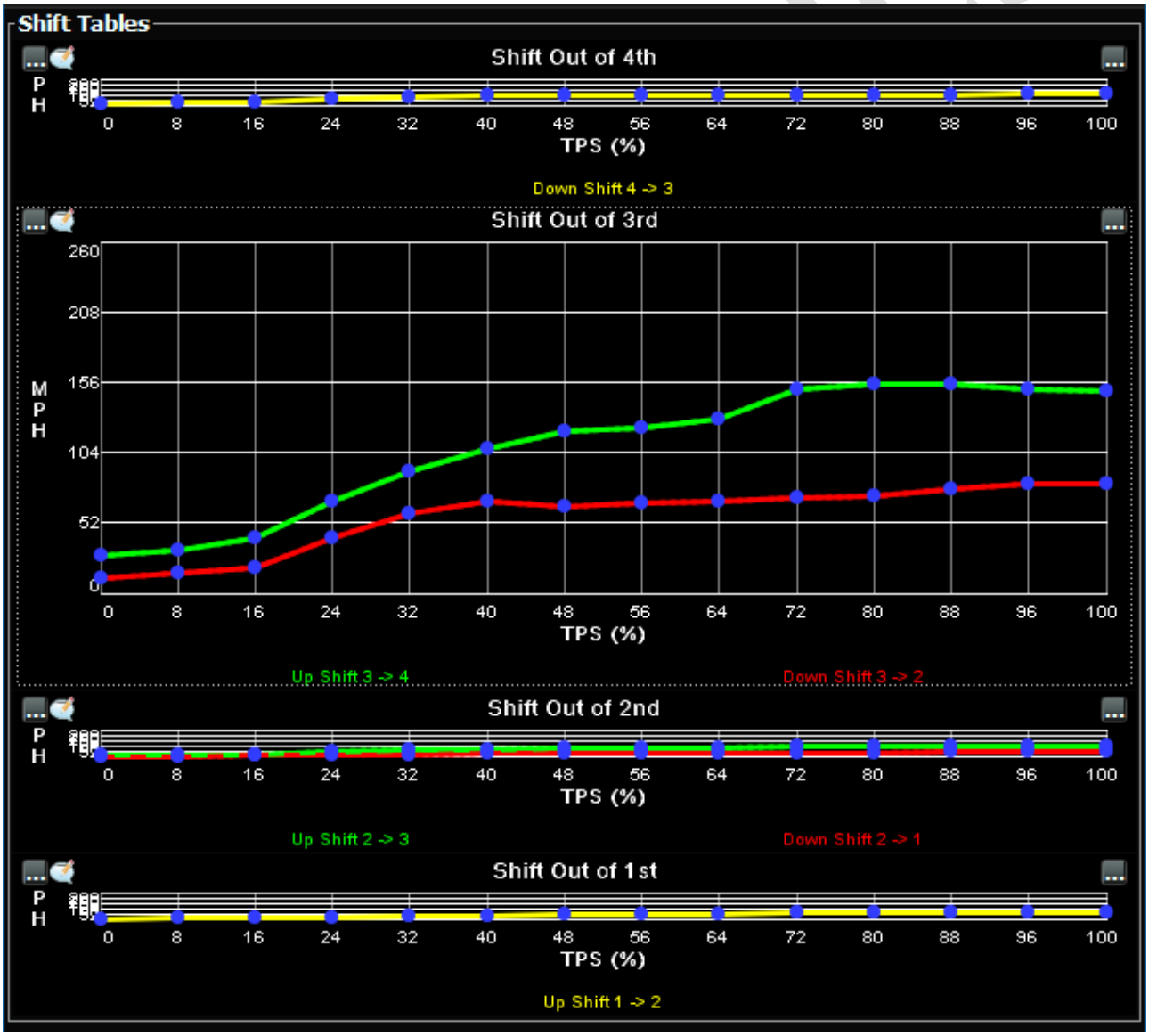


Figure 15-17

Code Snippet:

```
dialog = shiftTblsIndex_4sp, "Shift Tables", indexCard
    panel = shift_4spd
    panel = shift_out_3_curve,
    panel = shift_out_2_curve,
    panel = shift_out_1_curve,
```

The panels shift_4spd, shift_out_3_curve, shift_out_2_curve, shift_out_1_curve are all separate Curve Editors defined in the Curve Editor Section.

15.5 Dialog Examples

In this section will provide screenshots and the snippet required to construct the dialog. As these are snippets, in some cases there will be references to Constants and OuputChannels that are not defined here, but were in the original ini file. We will make note of the missing components and provide a brief explanation of their definitions.

15.5.1 Basic Dialogs

Starting with a basic dialog containing only scalar and bit parameter class Constants:

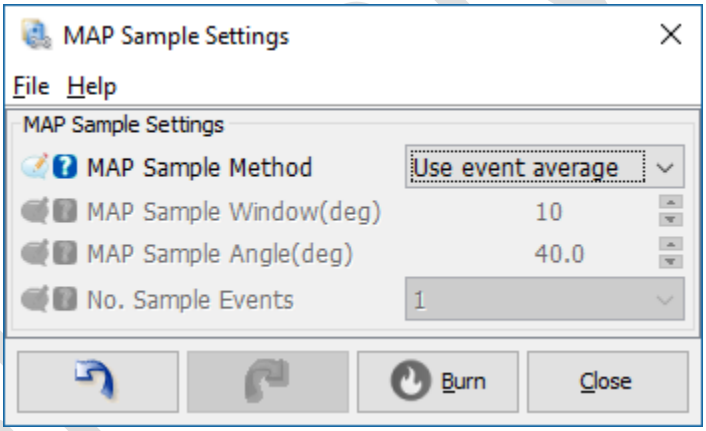


Figure 15-18

15.5.1.1 Basic Dialog Definition

```
dialog = map_sample_dialog, "MAP Sample Settings"
    topicHelp = "file://$getProjectsDirPath()/docs/Help.3.4.pdf#mapsamp"
    field = "MAP Sample Method", mapsample_opt2
    field = "MAP Sample Window",     mapsample_window, { mapsample_opt2 }
    field = "MAP Sample Angle", mapsample_angle, { mapsample_opt2 }
    field = "No. Sample Events", mapsample_opt1, { mapsample_opt2 }
```

Where:

mapsample_opt2 – A bit Constant defined in the [Constants] section
mapsample_window – A scalar Constant defined in the [Constants] section
mapsample_angle – A scalar Constant defined in the [Constants] section
mapsample_opt1 – A bit Constant defined in the [Constants] section

15.5.1.2 Visibility

The visibility of any row can be controlled within the ECU definition using an expression.

By editing the above dialog to add a conditional visibility expression:

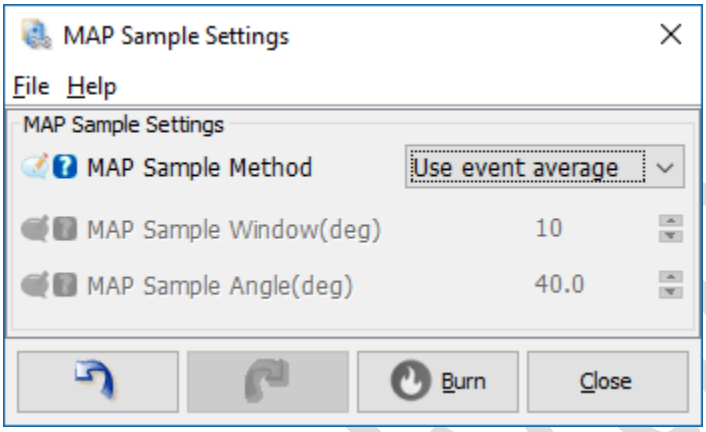


Figure 15-19

```
dialog = map_sample_dialog, "MAP Sample Settings"
    topicHelp = "file://$getProjectsDirPath()/docs/Help.3.4.pdf#mapsamp"
    field = "MAP Sample Method", mapsample_opt2
    field = "MAP Sample Window",     mapsample_window, { !mapsample_opt2 }
    field = "MAP Sample Angle", mapsample_angle, { mapsample_opt2 }
    field = "No. Sample Events", mapsample_opt1, { mapsample_opt2 }, {mapsample_opt2}
```

Similar to the Enabled Condition, the addition of the visibility expression changes the result of mapsample_opt2 being false so the field is not just disabled, but no longer visible.

15.5.2 Multi-panel Example

Using an example from the Layouts section, this dialog is actually a compound of 3 dialogs. 2 Dialogs are defined independently as the Left dialog, then a separate dialog is defined using a rad layout so it can toggle from a Curve Editor to a Table Editor based on the user selection. All basic field components are placed on in the left dialog. The final 3rd dialog uses a border layout. This allows the non resizable components in a fixed space, but the Curve/Table are placed Center where they can fill any remaining space if the dialog is stretched larger.

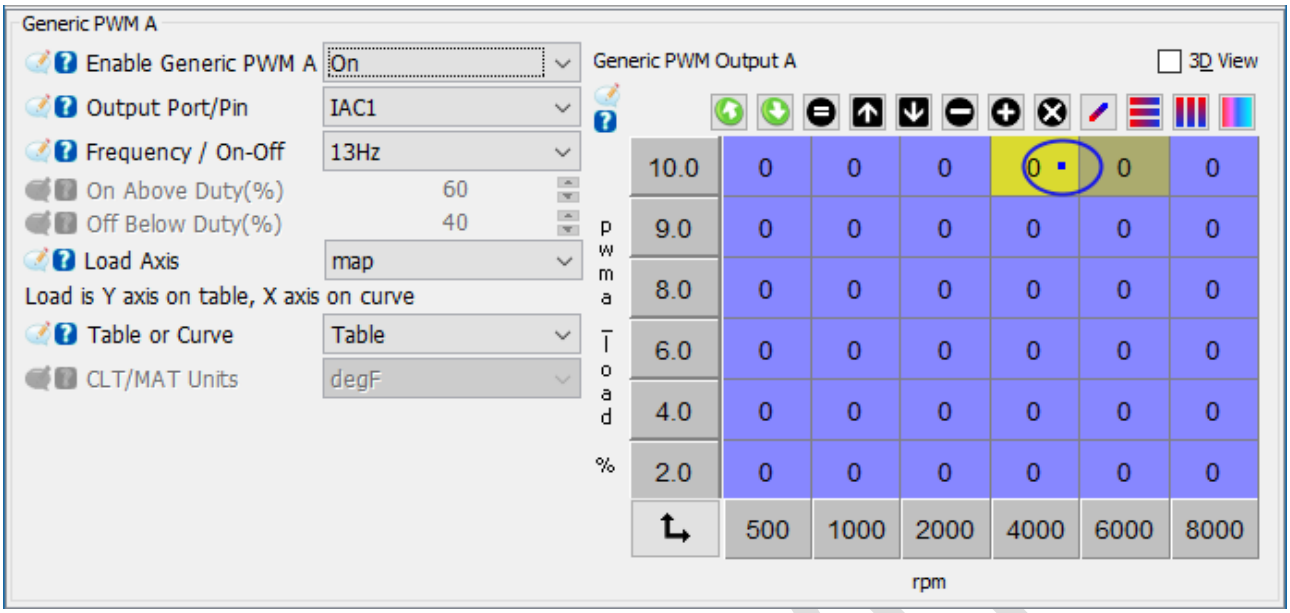


Figure 15-20

Syntax:

```
dialog = gen_pwm_left_a, ""
    field = "Enable Generic PWM A", pwm_opt_on_a
    field = "Output Port/Pin", pwm_opt2_a, {pwm_opt_on_a}
    field = "Frequency / On-Off", pwm_opt_freq_a, {pwm_opt_on_a}
    field = "On Above Duty", pwm_onabove_a, {pwm_opt_on_a && (pwm_opt_freq_a == 0)}
    field = "Off Below Duty", pwm_offbelow_a, {pwm_opt_on_a && (pwm_opt_freq_a == 0)}
    channelSelector= "Load Axis", pwm_load_offset, pwm_load_size, {pwm_opt_on_a}
    field = "Load is Y axis on table, X axis on curve"
    field = "Table or Curve", pwm_opt_curve_a, {pwm_opt_on_a}
    field = "CLT/MAT Units", sensor_temp, {0}
    field = "" ; filler
    field = ""
    field = ""
    field = ""
    field = ""
    field = ""

dialog = gen_pwm_curve_grapha, "", card
    panel = pwm_duties_Tbl_a, Center, {pwm_opt_on_a && (pwm_opt_curve_a == 0)}
    panel = pwm_curve_a, Center, {pwm_opt_on_a && pwm_opt_curve_a}

dialog = gen_pwm_a, "Generic PWM A", border
     topicHelp = "file://$getProjectsDirPath()/docs/ Reference-1.4.pdf#genpwm"
    panel = gen_pwm_left_a, West
    panel = gen_pwm_curve_grapha, Center
```

16 [FRONTPAGE]

The front page section is used to define what gauges and indicators will be displayed on the default dashboard and in what order. You can define 1 – n gauges. Each gauge entry will be assigned a gauge template defined in the [GaugeConfigurations] section

```
gauge1 = gaugeTemplate1
gauge2 = gaugeTemplate2
gauge3 = gaugeTemplate3
gauge4 = gaugeTemplate4
…
…
```

They will be laid out in 2 rows left to right, row by row. The common number of gauges is 8 or 10 for proper screen fitment.

1	2	3	4
5	6	7	8

Or for a 10 gauge layout:

1	2	3	4	5
6	7	8	9	10

Etc…

Alternatively, the application can use a Dashboard template with gauges each having an id assigned of numbers 1-n. In this case the list of gauge templates will be applied in that order when loading the default dashboard. This allows the use of a pre-designed dashboard that can be used with various firmwares with different gauge configurations.

Indicators are a visual reference to a Boolean condition. They are either on or off. The default rendering of this is a rectangle box with a label text. Each indicator is assigned:

- OutputChannel – Drives the state of the indicator, if value == 0 the state is Off, otherwise on
- Off Label – Text to be displayed when in off state, can include String Functions
- On Label – Text to be displayed when in on state, can include String Functions
- Off Background Color – Color of the background in an off state
- Off Text Color – Color of the label text in an off state
- On Background Color – Color of the background in an on state
- On Text Color – Color of the label text in an on state
- Enable Expression – Optional; If true the indicator will be included on the default dash, if false it will be present on the right click menu, but will not be on the default dashboard.

indicatorTemplate definitions are the same as indicators, but the key difference is they will be displayed on the right click menu, but not on the default dashboard.

16.1 EXAMPLE FRONTPAGE

```
       gauge1 = tachometer
       gauge2 = throttleGauge
       gauge3 = pulseWidth1Gauge
       gauge4 = cltGauge
       gauge5 = advdegGauge
       gauge6 = fuelloadGauge
       gauge7 = afr1Gauge
       gauge8 = matGauge
       gauge9 = pwGauge
       gauge10= dcGauge

;-------------------------------------------------------------------------------
;Indicators = OutputChannel  off-label  on-label,  off-bg,  off-fg,  on-bg, on-fg
;----------  -------------   ---------  --------   -------  -------  ------ -----
 indicator = { ready }, "Not Ready",      "Ready",   white, black, green, black
 indicator = { crank }, "Not Cranking", "Cranking",  white, black, green, black
 indicator = { startw }, "ASE OFF",      "ASE",      white, black, green, black
 indicator = { warmup }, "WUE OFF",     "WUE",       white, black, green, black
 indicator = { tpsaccaen}, "TPS Accel", "TPS Accel", white, black, green, black
 indicator = { mapaccaen}, "MAP Accel", "MAP Accel", white, black, green, black
 indicator = { tpsaccden }, "TPS Decel","TPS Decel", white, black, green, black
 indicator = { mapaccden}, "MAP Decel", "MAP Decel", white, black, green, black


 ; To be displayed on the right click menu, but not on the default dashboard

 indicatorTemplate = { accActive}, "ACC On", "ACC Off", white, black, green, black
```

*Each OutputChannel must be defined in the [OutputChannels] section.

17 [KEYACTIONS]

At this time there is only 1 KeyAction, showPanel.

Usage:

The showPanel Key Action allows you to define 1-n keys that if pressed in order and not released, will show the referenced dialog. This can be used in lieu of a Menu to launch a dialog allowing for hidden dialogs only to be shown with the proper key combination. If the dialog is password protected, the password protection is honored and the user will be required to enter the password to open the dialog.

```
    showPanel = keyCombination,  dialogName
```

Example:

```
  [KeyActions]
        showPanel = xyz,  myDialog
```

The result of this is if the user now presses and holds x then y then z, myDialog will open.

18 [LoggerDefinition]

18.1 Overview

There is support for extremely high speed memory logging to capture real-time firmware data at speeds that typical transports cannot support. This is accomplished through "batch logging". The controller is will assign a region of RAM, typically a page. The firmware then constructs compact data set records that are written to the allocated RAM as quickly as is needed or it can. Once the memory region is full, it pauses logging and notifies the application to read page. The firmware will resume logging after the log data read is completed. This type of logging works well for functions such as capturing trigger wheel patterns for diagnostics, capturing MAP signal in relationship to crank position or injector and igniter state related to crank angle.

The application is then responsible for reconstructing the data records into human readable data. Due to the limited RAM, sub byte sized fields and interleaved fields are commonly used to maximize the information captured within the limited RAM allocated.

The LoggerDefinition section allows the definition of 1-n Loggers with the commands and field definition for each.

18.1.1 loggerDef

Each new logger will begin definition with a loggerDef entry. All rows following are assumed be attributes of this logger entry until a new loggerDef entry is found or section end is reached.

18.1.1.1 Syntax:

```
;loggerDef = uniqueName, Display Name, type
loggerDef = mapLogger, "MAP Logger", csv
```

18.1.1.1.1 Attributes

uniqueName – the reference name for this logger definition.

Display Name – The name of this logger display to the user and in the Logger selection.

Type – this drives the graphic viewer used to render the data to the user. The valid types are:

- composite – Renders crank and cam teeth with sync errors in an oscilloscope style.*
- tooth – Renders a bar chart style with each bar representing the time between crank teeth.*
- trigger – Renders a bar chart style with each bar representing the time between trigger events.*
- cvs – Renders and X-Y Plot. This is a generic format that works with any data set.

* A specific data format required.

18.1.2 Commands

startCommand – The command to start the logger. Standard Request reply protocol format.

stopCommand – The command to stop the logger. Standard Request reply protocol format.

dataReadCommand – The command to read the controller RAM. Standard Request reply protocol format.

dataReadyCondition – An expression to that will resolve to true where the controller RAM is ready to be read.

dataReadTimeout – The maximum time in ms before the application should wait for the dataReadyCondition to become true before reading the controller RAM.

continuousRead – Boolean flag. If false, the application only expects 1 RAM read, then stop. Otherwise it will continuously wait for the next page of data.

dataLength – The number of bytes expected for each read.

recordDef – Defines the header length, footer length and record length in bytes.

Format:

```
; recordDef = headerLen, footerLen, recordLen
```

Header and footer are optional, recordLen is required

defaultXAxis – Will set the default field for the UI to use in a X-Y plot of the Logger data.
Example:

defaultXAxis = "A defined Field"

verticalMarker – Will place a vertical marker for reference on the X-Y Chart.
Format:

;verticalMarker = "Y Field Name", "X Field Name", "Title", position
verticalMarker = "Pressure", "Crank Angle", "0° TDC", 0

overlaidDatasetCount – Sets the number of data sets to combine to remove NaN columns. This can be used to deliver each column in a separate block, then NaN values will be replaced by a subsequent block of data.

logProcessorType - - By default the data will be read directly from the connection that is currently being used for communication with the controller. However, in the case you require very high speeds and and have an Ethernet connection and UDP capabilities, you can have the data broadcast using a UDP Stream. To enable this set the logProcessorType and port to listen on. The control commands will still be sent via the typical connection. See the example definition for a cylinder pressure logger at the end of this section:

logProcessorType = UDP_Stream
slavePort = 25555

stopOnExit – When set to true, the logger will be stopped upon exiting the view. In the case of TunerStudio that would be upon leaving the High Speed Loggers tab. Default; false

18.1.3 Field Definitions

There are 3 types of fields that can be defined.

headerField – An optional field directly from a set of bits in the header with scale applied.

recordField – A field directly from a set of bits in the record with scale applied.

calcfield – A field that is calculated from recordField and Constants.

18.1.3.1 headerField

A header for each read can optionally be included. If defined, the header allows the additional fields to be defined for the read for values that can be considered consistent through the rest of the read. This can save space in the record for values that will not change. There is only 1 header per read. Addressing is in bit positions.

Format:

```
;headerField = Name,  HeaderName, startBit, bitCount, scale, units
```

Name – reference name of this headerField. To be referenced for any calcFields

HeaderName – The header for the log file.

startBit – position of 1st bit in the record for this field.

bitCount – the number of bits used for this field.

scale – the scale to convert from raw value to user value.

units – The units for the log file header.

18.1.3.2 recordField

A field directly from a set of bits in the record with scale applied. All addressing on the record is in bits as opposed to the more typical bytes.

Format:

```
recordField = Name,     HeaderName,  startBit, bitCount, scale, units, updateCondition
```

Name – reference name of this recordField. To be referenced for any calcFields

HeaderName – The header for the log file.

startBit – position of 1st bit in the record for this field.

bitCount – the number of bits used for this field.

scale – the scale to convert from raw value to user value.

units – The units for the log file header.

updateCondition – Optional. For interleaved fields, a condition can be set to update the value of this field. If false, it will retain the last updated value. This allows record data to be multi-mapped to prevent redundant data. If not defined, the field is assumed to be updated with each record.

18.1.4 Additional Attributes

defaultXAxis – Will set the default field for the UI to use in a X-Y plot of the Logger data.
Example:

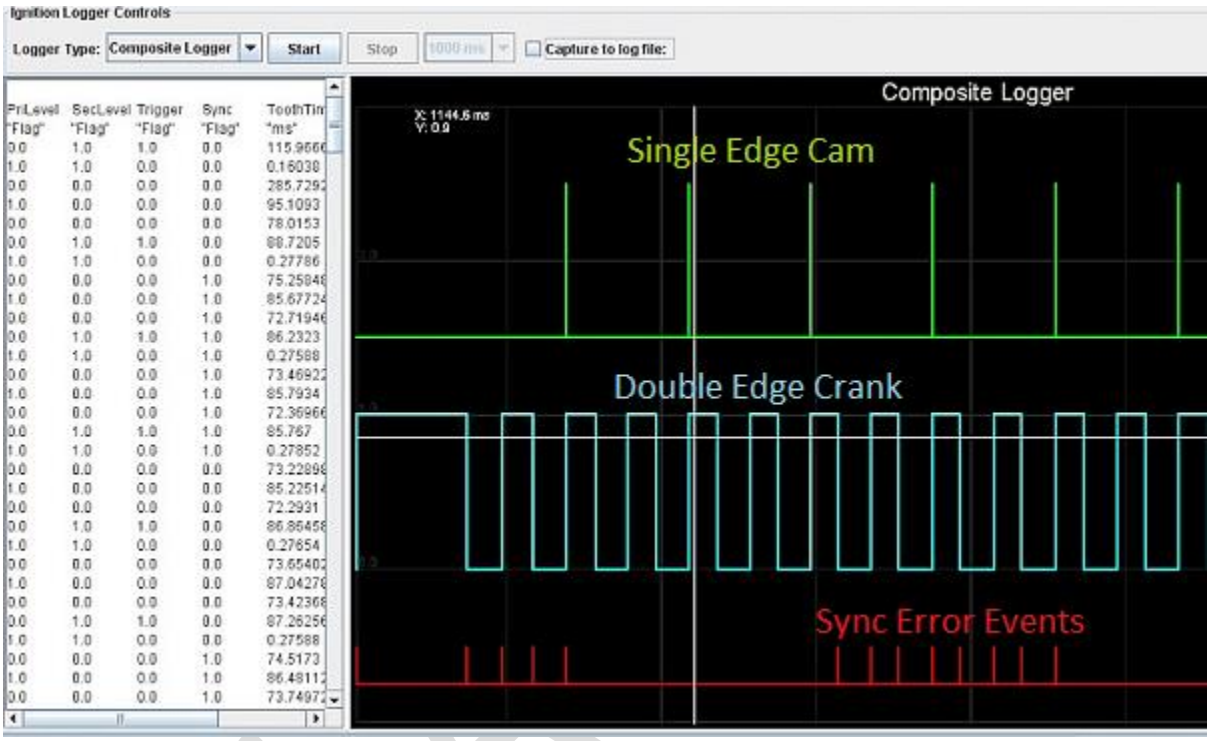
```
defaultXAxis = "A defined Field"
```

verticalMarker – Provides vertical marker points for the UI to place along the X A
verticalMarker = "Y Field Name", "X Field Name", "Title", position

18.2 EXAMPLE LOGGERDEFINTIONS

18.2.1 composite log

The composite log renders an oscilloscope “like” view for displaying the wheel teeth as the ECU sees it.



18.2.1.1 Required Log fields:

In order for the application to render the data using the Composite logger view, specific key fields are required in the captured log.

PriLevel – The current level of the primary wheel, typically the crank tooth level.
SecLevel – The level of the secondary wheel, typically the cam tool level.
Trigger – Which wheel created the even, primary=0, secondary=1
ToothTime – Time in ms since last event.
Time – Total running time in ms.

18.2.1.1.1 Example defintion

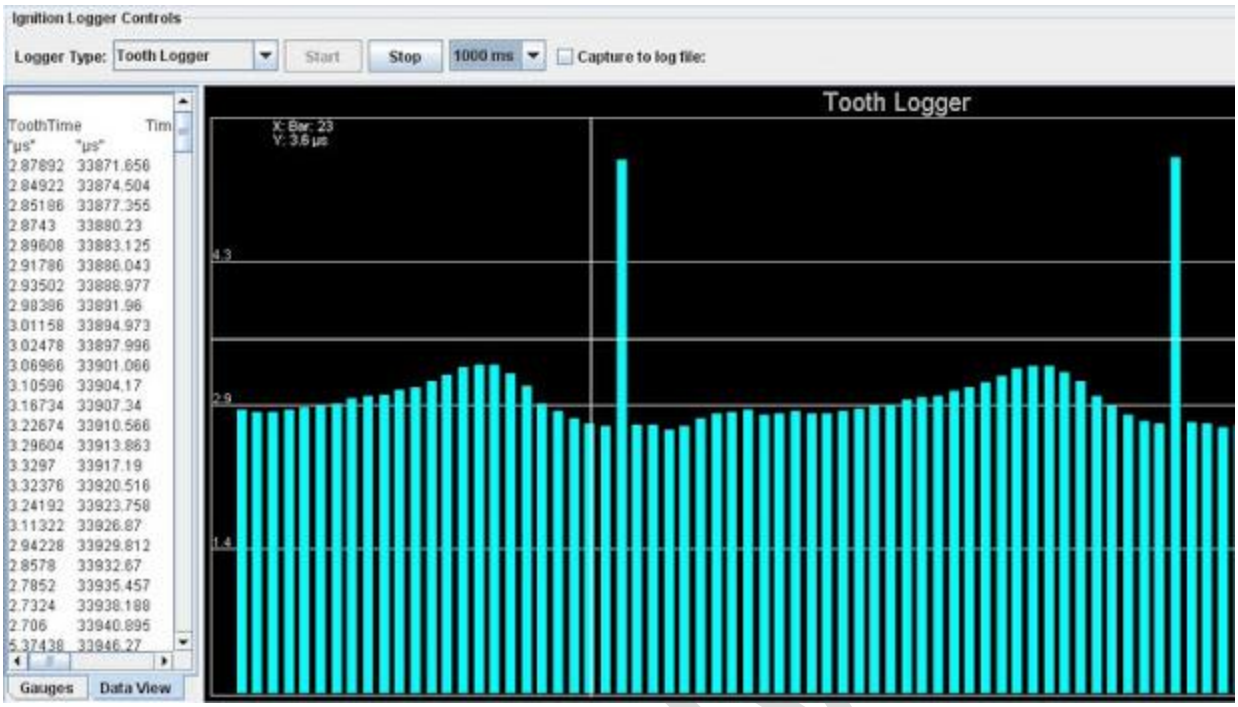
```
loggerDef = compositeLogger, "Composite Logger", composite

   startCommand = "w\$tsCanId\x05\x00\x0A\x00\x01\x01"
   stopCommand = "w\$tsCanId\x05\x00\x0A\x00\x01\x03"
   dataReadCommand = "r\$tsCanId\x08\x00\x00\x08\x00" ; 2K, std TS command format
```

```
dataReadTimeout = 60000 ; time in ms
dataReadyCondition = { logStat == 2 }
dataLength = 2048 ; in bytes, including headers, footers and data //not used..
continuousRead = false ;
recordDef =   0,   0,   5; in bytes, the recordLen is for each record

;recordDef = headerLen. footerLen, recordLen

;recordField = Name,          HeaderName, startBit, bitCount, scale, units, updateCondition
recordField = sync,           "Sync",             36,        4,     1.0,   ""
recordField = crnkCam,       "CrnkCm",            34,        2,     1.0,   ""
recordField = edge,            "Edge",            32,        2,     1.0,   ""
recordField = refTime,      "RefTime",             0,       32,     0.001,   "ms"


; hidden calcField serves as intermediate variable
calcField = maxTime,               "MaxTime",   "ms",  { maxValue(refTime) }, hidden

; Composite compatability fields
recordField = priLevel,    "PriLevel",   32,        2,     1.0,   "H/L", { crnkCam == 0 }
recordField = secLevel,    "SecLevel",   32,        2,     1.0,   "H/L", { crnkCam == 1 }
recordField = trigger,     "Trigger",    34,        2,     1.0,   ""

calcField = toothTime,     "ToothTime",   "ms",  { refTime - pastValue(refTime, 1) }

; currently the "Time" field needs to be at the end of the row for it to jump to the
; record when clicking on the chart.
; So this is created here
calcField = time,              "Time",   "ms",  { refTime }
```

18.2.1.2 Tooth Logger

Renders a bar chart with tooth times for a visual view of tooth times to aid in seeing inversed polarity or dropouts in input signal.

This log type requires the fields:
ToothTime – The time since last tooth in



18.2.1.3 cvs Logger

This is the most common and flexible logger. There are no constraints on what fields are logged. The graphical rendering will be an X-Y plot supporting a single X axis with multiple Y axis traces. These logs are often well viewed in MegaLogViewer using scatter plots or in the TunerStudio integrated X-Y chart.

Here we will define a Cylinder Pressure logger using UDP Streams. In this example the flexibility of how your data may be passed is demonstrated at the cost of showing how complex it can get.

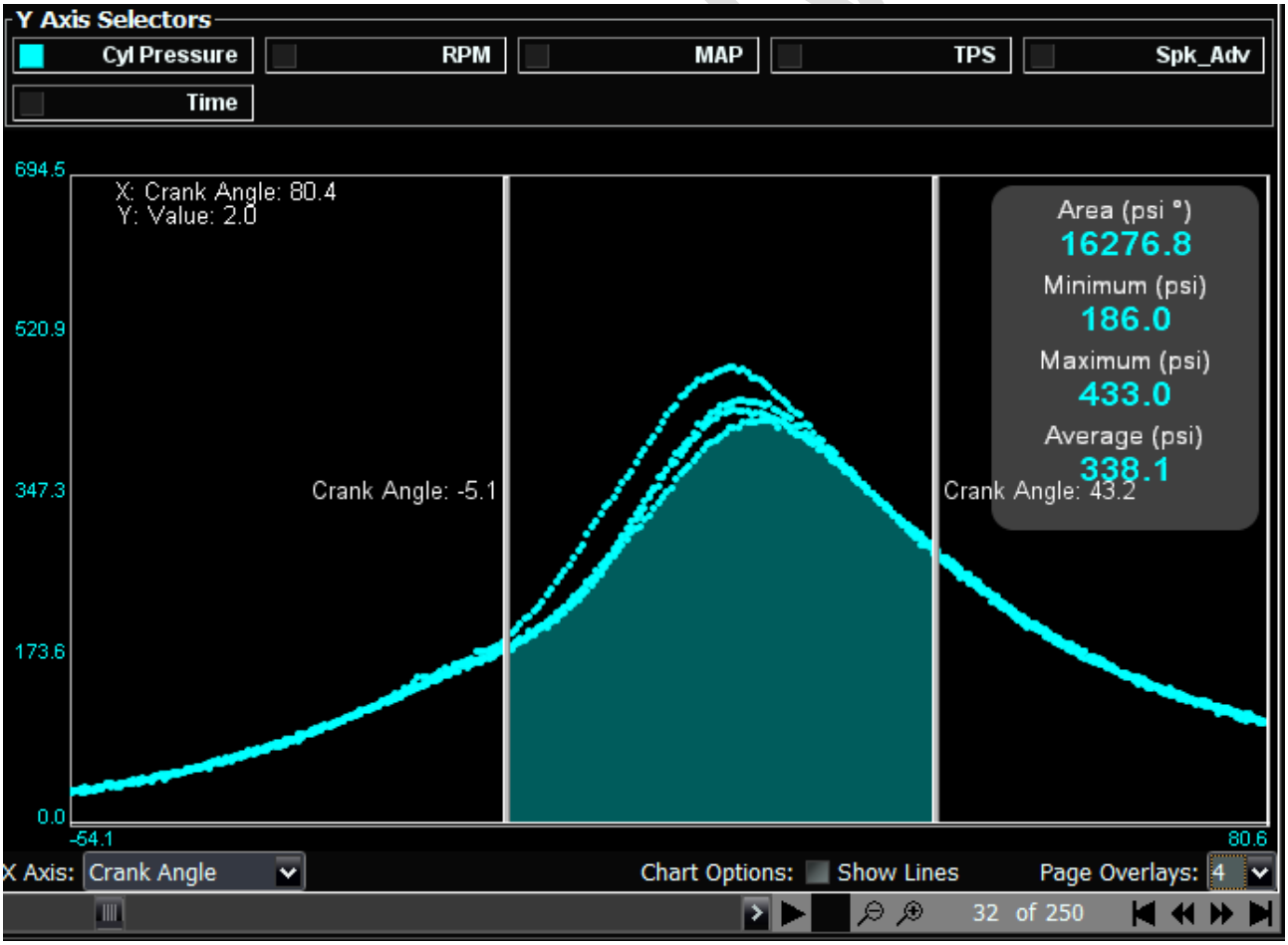
How this data is structured. Each UDP datagram contains the cylinder pressure for a pair of cylinders from 135° BTDC to 135° ATDC every 1 crankshaft degree. Thus a UDP datagram contains 270 4 byte records with 2 U16's in each record representing the cylinder pressure for 2 cylinders. This makes a total of 1080 bytes per datagram. The overlaidDatasetCount attribute is used to accumulate the data from 4 datagrams to include the pressures from all 8 cylinders in each single page of data displayed. The defined header field pressurePair allows the software to associate each datagram with the pair of specific cylinders.

The firing order is defined elsewhere in the tune and the logs header labels are generated using $stringValue() function referencing the firing order array in the tune data. The

```
loggerDef = cylPressLogger, "Cylinder Pressure Logger", csv
   startCommand = "\xFF\x00"
   stopCommand = "\xFE"
   ;continuousRead = true ;
   logProcessorType = UDP_Stream
   slavePort = 25555

   ;recordDef = headerLen. footerLen, recordLen
   recordDef =   4,   0,   4; in bytes, the recordLen is for each record (multiple channel)

   overlaidDatasetCount = 4;
```

```
;headerField = NBytes,           "No. Bytes",      0,      32,    1.0,      ""
headerField = pressurePair,  "Pressure Pair",      30,      2,      1,      "", {1},  hidden
headerField = cntr,               "Counter",       0,      16,      1,      "", {1},  hidden

;recordField = Name,   HeaderName,   startBit, bitCount, scale, translate, units, updateCondition
recordField = cylPressA, "Cyl $stringValue(Fire_Order_a)", 16, 16, {GAIN}, "psi", { pressureCyl == 0}
recordField = cylPressB, "Cyl $stringValue(Fire_Order_b)", 16, 16, {GAIN}, "psi", { pressureCyl == 1}
recordField = cylPressC, "Cyl $stringValue(Fire_Order_c)", 16, 16, {GAIN}, "psi", { pressureCyl == 2}
recordField = cylPressD, "Cyl $stringValue(Fire_Order_d)", 16, 16, {GAIN}, "psi", { pressureCyl == 3}
recordField = cylPressE, "Cyl $stringValue(Fire_Order_e)", 0,  16, {GAIN}, "psi", { pressureCyl == 0}
recordField = cylPressF, "Cyl $stringValue(Fire_Order_f)", 0,  16, {GAIN}, "psi", { pressureCyl == 1}
recordField = cylPressG, "Cyl $stringValue(Fire_Order_g)", 0,  16, {GAIN}, "psi", { pressureCyl == 2}
recordField = cylPressH, "Cyl $stringValue(Fire_Order_h)", 0,  16, {GAIN}, "psi", { pressureCyl == 3}
calcField = startAngle,     "StartAngle",   "°",  -135, hidden

calcField = crankAngle,     "Crank Angle",   "°", { startAngle + (highSpeedRecordNumber()+1)}

;verticalMarker = "Y Field Name", "X Field Name", "Title", position
verticalMarker = "Crank Angle", "Crank Angle", "0° TDC", 0

defaultXAxis = "Crank Angle"
```

Example rendering of this data:



19 [PORTEDITOR]

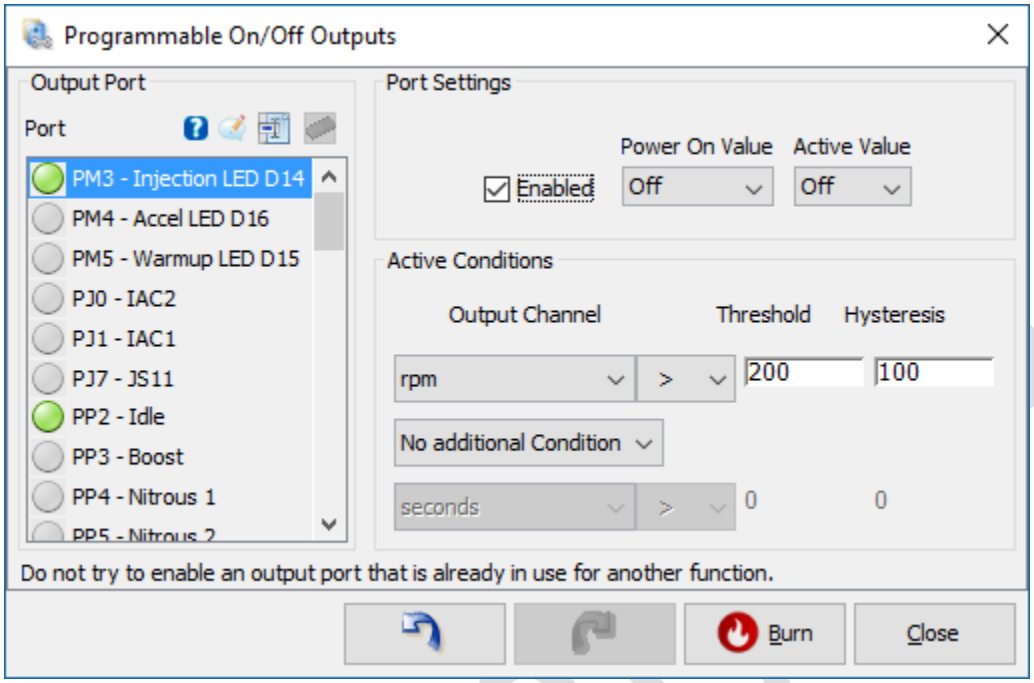


Figure 19-1

The PortEditor is a configurable standard dialog. It allows the creation of trigger condition for the Controller to activate and deactivate specified outputs based on user defined conditions. By mapping each keyword to an appropriate array, the UI will then set the array values based on user input.

The configuration of this dialog involves a 1D array of a length equaling the number of output pins, then a series of arrays nPins x nOfconditions. In Figure 19-1 there 12 Pins defined with 2 possible conditions, thus each condition array is expected to be [12x2] in size. If you wish to have 3 conditions, resize these arrays to [12x3] each.

Array Configuration for 12 PINs and 2 conditions:

1D Arrays:

- enabledPorts – Map to a 1D array number of PINS (12) long. 1=enabled, 0=disabled.
- powerOnValue – Pin State when controller boots. 0=off 1=on
- triggerValue – Pin State when user condition met. 0=off 1=on

2D Arrays size size [numberOfPorts x NumberOfConditions-1]:

- conditionalRelationship – array [numberOfPorts x nConditions – 1] Set relation between conditions

2D Arrays, each of size [numberOfPorts x NumberOfConditions]:

- outputCanId (optional) – Will hold the CAN ID of the target controller.
- outputOffset – The runtime read offset of the selected controller [OutputChannels]/ When using XCP, this will contain a 4 byte ODT address.

- outputName (optional)– Holds a CRC16 of the OutputChannel name. When the address for the referenced channel does not match, the address/offset will be updated. This array needs to be of type U16.
- outputSize – The size in bytes of the selected controller [OutputChannels], if extendedDataInSize option is active, upper 2 bits will indicate floating point or sign types. 0x40 set represents a Floating point, 0x80 represents a Signed type.
- operators – The select operational relationship and permitted operators
- threshold – Set user threshold value, this will be scaled as the [OutputChannels] selected.
- hysteresis – Set user hysteresis, will be scaled to match the selected [OutputChannels]
- portEnabledCondition (optional) – Hold array of expressions, 1 for each port label to determine if the PortLabel should be visible. These must be in the same index order as the defined Port Labels.
- portActiveDelay(optional) – An array with the time delay before the port will become active, after the condition is true.
- portInactiveDelay(optional) – An array with the time delay before the port will become inctive, after the condition is true.

```
 [PortEditor]
    ; map the arrays and supply the labels.
    ; all arrays are expected to be the same length in the first dimension
    ; thus a [ 7] array will be 7 ports in length and expect 7 labels
    ; The second dimension will drive the number of conditions per array.
    ; thus a [7x2] array will have 2 conditions joined by the conditionRelationship.
    ; a [7x3] will have up to 3 conditions.
    portEditor = std_port_edit, "Output port Settings"
        topicHelp = "http://www.megamanual.com/mt28.htm#sp"
        ; 1st the array constant, then the labels in the index order.
        enabledPorts = psEnabled, "PM2 - FIdle", "PM3 - Injection LED", "PM4 - Accel
LED", "PM5 - Warmup LED", "PT6 - IAC1", "PT7 - IAC2", "PA0 - Knock Enable"
        ; new CAN id, optional if psCanId is set to a valid array equal in size to
                        ;outputOffset, it will be in the UI.
        outputCanId  = psCanId ; optional CAN ID of the target controller.
        outputOffset = psOutOffset
        outputSize   = psOutSize
        operators    = psCondition, "<", "=", ">" ; the actual ASCII value of the
                       ; operator will be sent to the controller
        threshold    = psThreshold
        hysteresis   = psHysteresis
        powerOnValue = psInitValue
        triggerValue = psPortValue
        conditionRelationship = psConnector, " ", "|", "&"
        portEnabledCondition = { 1 }, { NOS_STAGES_RACE < 1 }, { NOS_STAGES_RACE < 2 }, {
(NOS_STAGES_RACE < 3) }, { 1 }, { 1 }, { 1 };
        activateOption = extendedDataInSize ; Will place sign and type flags in upper
nibble of outputSize
```

The appropriate arrays should be defined in [Constants] section. In the above example these Constants are assumed to be defined.

- psEnabled – [7]
- psOutOffset – [7x2]
- psOutSize – [7x2]

- psCondition – [7x2]
- psThreshold – [7x2]
- psHysteresis – [7x2]
- psInitValue – [7]
- psPortValue – [7]
- psConnector – [7]
- psCanId – [7x2]

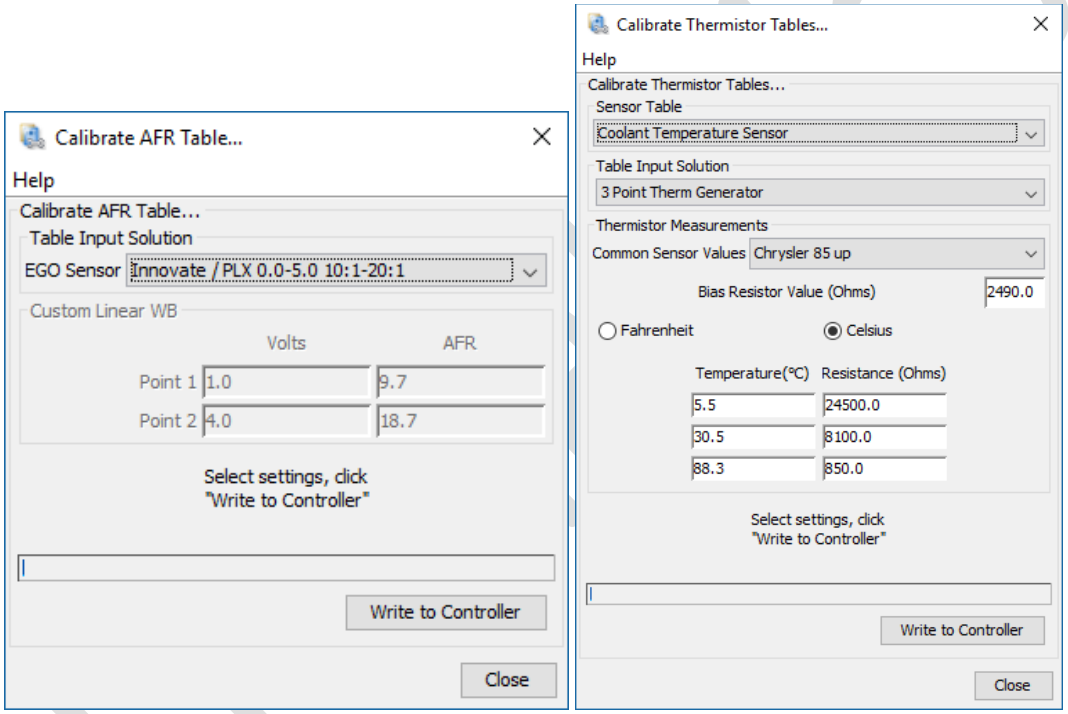
```
Example Constants section:
      psEnabled       = array ,  U08,      0,    [   51], "",  1, 0,      0,     1, 0, noSizeMutation
      psCondition     = array ,  U08,     51,    [ 51x2], "",  1, 0,     32,   124, 0, noSizeMutation
      psConnector     = array ,  U08,    153,    [   51], "",  1, 0,     32,   124, 0, noSizeMutation
      psInitValue     = array ,  U08,    204,    [   51], "",  1, 0       0,     1, 0, noSizeMutation
      psPortValue     = array ,  U08,    255,    [   51], "",  1, 0,      0,     1, 0, noSizeMutation
      psOutSize       = array ,  U08,    306,    [ 51x2], "",  1, 0,      0,   255, 0, noSizeMutation
      psOutOffset     = array ,  U16,    408,    [ 51x2], "",  1, 0,      0,  1024, 0, noSizeMutation
      psThreshold     = array ,  S16,    612,    [ 51x2], "",  1, 0, -32768, 32767, 0, noSizeMutation
      psHysteresis    = array ,  S16,    816,    [ 51x2], "",  1, 0, -32768, 32767, 0, noSizeMutation
```

20 [ReferenceTables]

20.1 Overview

Sensor data is commonly non-linear creating a situation where the most efficient way to convert adc input data to a true value becomes a lookup table. The [ReferenceTables] section enables text based definition of raw lookup tables that can be scaled and written to the controller as normalized data. This allows the use of a broad set of supported sensors and the ability to support new sensors without a firmware change. In this section you can define multiple referenceTables and tableGenerators.

The view of this component will be driven by the selected generator and solution



20.2 Commands

tableWriteCommand – the write command for writing the table. Standard Request reply protocol format.

tableBlockingFactor – The maximum block size that will be written during table writes.

tableCrcCommand – Optional CRC32 command. This is used to verify the table in the controller matches a known solution.

20.3 Defining a Reference Table

Reference table definitions will always start with a row beginning with the keyword referenceTable and attributes name and title:

```
referenceTable = std_therm_clt, "Calibrate Coolant Table..."
```

Additional attributes that will be defined in the rows immediately following.

20.3.1 Required Attributes

tableIdentifier – a single byte numeric identifier to specify the target table. Multiple tables can be added to a single dialog using a comma delimited string of id's and titles:

```
tableIdentifier  = 0, "Coolant Temperature Sensor",  1, "Air Temperature Sensor"
```

adcCount – the number of translated elements in the table.

bytesPerAdc– the number of bytes of the translated element for each ADC, 1 for U08, 2 for U16, etc.

scale – The amount to scale the value by before writing it to the controller. Using 10, 14.7 -> 147

20.3.2 Optional attributes:

tableStartOffset – set the offset to the 1st write position. Default is 0

tableLimits – specifies the min and max values to write to the table, with the default value for the railed out values.

```
                  id,   min, max, default
tableLimits = 000, -40, 350, 180
```

min = -40
max = 350°F
default = 180°F

topicHelp – Optional help reference. See User Help

20.3.3 Solutions

Each Reference table is assigned 1 or more solutions. A solution will have 1 or more fully configured Table Generators or any expression that will produce the desired value through mathematical expression or lookup using an inc file.

20.3.4 Table Generators

The Table Generator will be responsible for producing the raw data to be written to the controller. The user inputs are up to the generator selected.

Each reference table must have at least 1 table generator assigned.

Available Table Generators:

thermGenerator - as used for temp sensor calibration

linearGenerator – will generate linear output based on 2 volt / AFR points.

fileBrowseGenerator – will allow the user to browse for their own inc file for custom sensors

20.3.4.1 thermGenerator

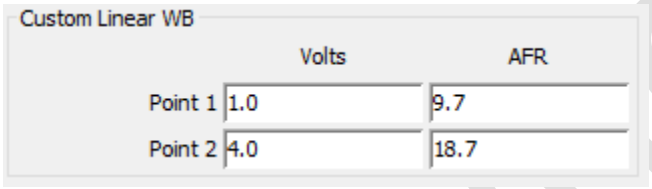
The thermGenerator is primarily for temperature sensors. It allows the user to input 3 resistance / temperature points, then generates the non-linear standard temperature ADC data. You may also add 1-n pick list options.

To assign pick list options the keyword thermOption is used:

```
tableGenerator  = thermGenerator, "Thermistor Measurements"
;thermOption = name, resistor bias, tempPoint1(C), resPoint1, tempPoint2, resPoint2, tempPoint3, resPoint3
      thermOption     = "GM",     2490,    -40, 100700,   30, 2238,     99, 177
      thermOption     = "Chrysler 85 up",     2490,    5.5,24500, 30.5, 8100,    88.3, 850
      thermOption     = "Ford",    2490,     0, 94000,      50, 11000,    98, 2370
      …
```

20.3.4.2 linearGenerator

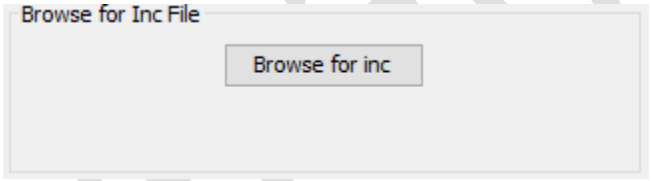
A 2-point generator typically used for linear WB definition if no picklist solution is available.




```
solution      = "Custom Linear WB",          linearGenerator
```

20.3.4.3 fileBrowseGenerator

A widget that will display a File Browse button, then generate the data from the inc file.

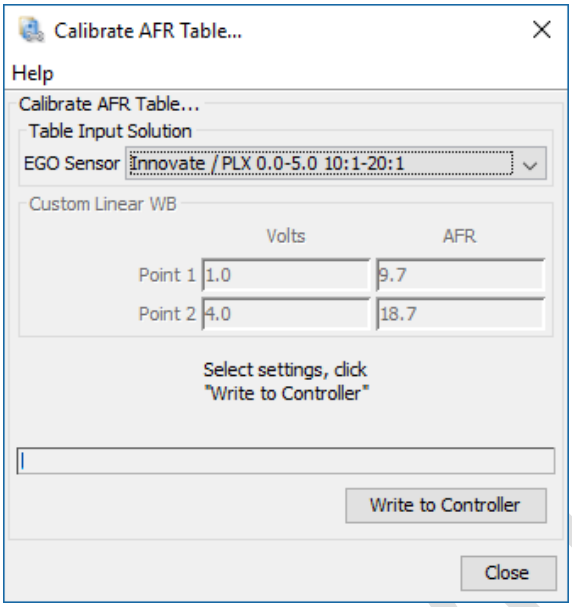



```
solution         = "Custom inc File",           fileBrowseGenerator
```

20.4 EXAMPLE ENTRIES:

20.4.1 O2 Calibration

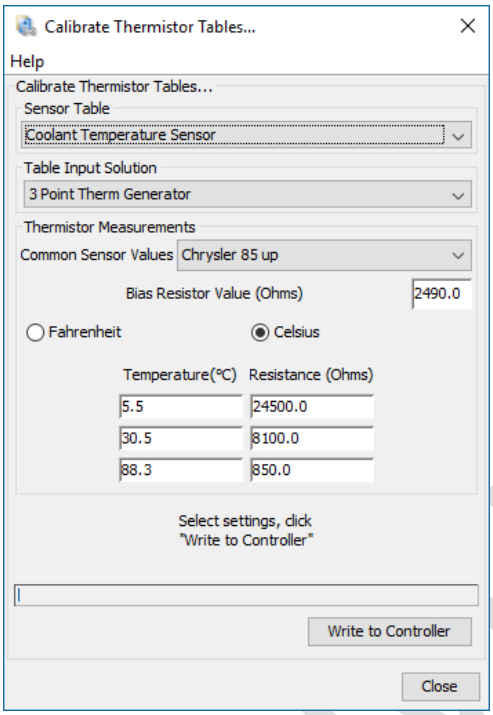
This entry will write a table of 1024 single byte values as AFR * 10




```
referenceTable = std_ms2geno2, "Calibrate AFR Table..."
   topicHelp = "http://www.megamanual.com/mt28.htm#oa"
   tableIdentifier = 002, "AFR Table"
   adcCount    = 1024  ; length of the table
   bytesPerAdc         = 1     ; using bytes
   scale               = 10 ; scale by 10 before sending to controller
   ;tableGenerator     = Generator Type, Label, xUnits, yUnits,  xLow, xHi,  yLow, yHi ;
   tableGenerator      = linearGenerator, "Custom Linear WB", "Volts","AFR",    1,   4,   9.7,  18.7
   tableGenerator  = fileBrowseGenerator, "Browse for Inc File"

   solutionsLabel      = "EGO Sensor"
   solution        = " ",                  { } ; bank row in case no match found. Must remain at top.
   solution    = "Narrowband",               { table(adcValue*5/1023 , "nb.inc") } ;
   solution = "AEM Linear AEM-30-42xx",      { 9.72 + (adcValue * 0.0096665) } ; 9.72:1 - 19.60:1
   solution = "AEM 30-2310, 30-4900, X-Series",     { 7.3125 + (adcValue * 0.0116080) } ; 7.31:1 - 19.18:1
   solution = "Autometer 0V=10:1, 4V=16:1", { 10 + (adcValue * 0.0073313783) }
   solution = "Daytona TwinTec",             { 10.01 + (adcValue    * 0.0097752) }
   solution = "DIY-WB",                      { table( adcValue/4, "WBlambda100MOT.inc")*14.7 / 100.0  }
   solution = "DynoJet Wideband Commander", { adcValue    * 0.00784325 + 10 }
   solution = "F.A.S.T. Wideband",           { adcValue    * 0.01357317 + 9.6 } ; 838.8608
   solution = "FJO WB",                      { table(adcValue*5/1023 , "fjoWB.inc" ) }
   solution = "Innovate LC-1 / LC-2 Default",       { 7.35 + (adcValue    * 0.01470186 )}
   solution = "Innovate / PLX 0.0-5.0 10:1-20:1", { 10 + (adcValue    * 0.0097752)}
   solution = "Innovate 1.0-2.0",            { adcValue * 0.049025}
   solution = "LambdaBoy",                   { table(adcValue*5/1023 , "lambdaBoy.inc" ) }
   solution = "NGK Powerdex",                { 9 + ( adcValue * 0.0068359375 ) }
   solution = "TechEdge DIY Non-Linear",     { table(adcValue*5/1023 , "TechEdge_DIYwbo2.inc") }
   solution = "TechEdge Linear",             { adcValue    * 0.0097752 + 9 }
   solution = "Zeitronix - Non Linear",      { table(adcValue*5/1023 , "zeitronix.inc") }
   solution = "Zeitronix - Linear Default", { 9.6 + (adcValue    * 0.0097752) }
   solution = "Custom Linear WB",          linearGenerator
   solution        = "Custom inc File",           fileBrowseGenerator
```

20.4.2 Temperature Sensor Calibration

Managing 2 reference tables. Each is 1024 WORD temperature values




```
referenceTable = std_ms2gentherm, "Calibrate Thermistor Tables..."
    topicHelp = "http://www.megamanual.com/mt28.htm#oh"
     tableIdentifier    = 000, "Coolant Temperature Sensor", 001, "Air Temperature Sensor"
        ; tableLimits (optional) = intentifier, min, max, defaultVal
        ; will set the default value if value is outside the min and max limits.
    tableLimits    = 001, -40, 350, 70
    tableLimits = 000, -40, 350, 180
    adcCount     = 1024     ; length of the table
    bytesPerAdc     = 2     ; using shorts
    scale        = 10     ; scale by 10 before sending to controller
    ;tableGenerator = Generator type, Label
    tableGenerator  = thermGenerator, "Thermistor Measurements"
    tableGenerator  = fileBrowseGenerator, "Browse for Inc File"
    thermOption     = "GM",     2490,    -40, 100700,     30, 2238,     99, 177
    thermOption     = "Chrysler 85 up",     2490,    5.5,24500, 30.5, 8100,    88.3, 850
    thermOption     = "Ford",     2490,      0, 94000,     50, 11000,    98, 2370
    thermOption     = "Saab (Bosch)",     2490,      0, 5800,     80, 320,    100, 180
    thermOption     = "Mazda",     50000,    -40, 2022088,     21, 68273,    99, 3715
    thermOption     = "Mitsu",     2490,    -40, 100490,     30, 1875,    99, 125
    thermOption     = "Toyota",    2490,    -40, 101890,     30, 2268,    99, 156
    thermOption     = "RX-7_CLT(S4 & S5)", 2490, -20, 16200, 20, 2500, 80, 300
    thermOption     = "RX-7_MAT", 42200, 20, 41500, 50, 11850, 85, 3500
    thermOption     = "RX-7_AFM(S5 in AFM)", 2490, -20, 16200, 20, 2500, 80, 300
    thermOption     = "BMW E30 325i",    2490,    -10, 9300,    20, 2500,     80, 335
    solution        = "3 Point Therm Generator",  thermGenerator
    solution        = "Custom inc File",    fileBrowseGenerator
```

21 [SettingContextHelp]

Use this section to add short help tips to each constant or PcVariable . List the constants you would like to add help tips to then type out the tip inside the quotes. The help tip will appear on the dialog as a small blue square with a

question mark which can be viewed by hoovering over the question mark or clicking on it to open a small dialog bubble. Exp. Below

[SettingContextHelp]

; constantName = "Help Text"

MatRtdRPMHi = "Full MAT spark retard is applied above this RPM."

MatRtdRPMLo = "No MAT spark retard is applied below the this RPM."

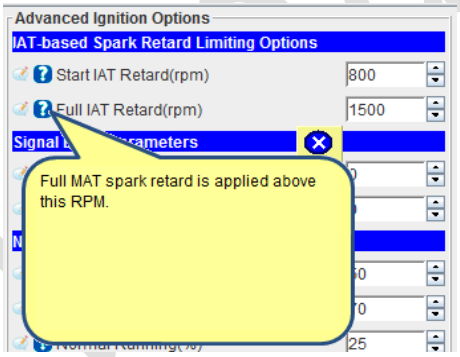


Figure 21-1

22 EXPRESSIONS AND MATH FUNCTIONS

22.1 EXPRESSIONS

Expressions are built using the values of other Constants, PcVariables or OutputChannels combined with supported functions and operators to produce values for each attribute. This allows the values to be dynamically set based on other conditions and preferences.

Expressions can be used in in most cases instead of hard numeric values. This is common with scale and translate

It is very acceptable to use expressions extensively throughout the ECU Definition file. Expressions are used to determine when components are enabled or disabled as well as visible or not. There is a fast and flexible math interpreter built into all applications. The syntax to be followed is c like expressions supporting all standard operators with proper order of execution.

For complex expressions, it is common to create a formula based OutputChannel that can then be referenced as a single variable. When creating an OutputChannel, you can reference any Constant or OutputChannel.

22.2 OPERATORS

• () grouping • ! logical NOT • ~ bitwise NOT • - unary minus = negation • * multiplication • / division • % modulus • + addition • - subtraction • << left shift • >> right shift • < less than	• <= less than or equal to • == equal to • > greater than • >= greater than or equal to • != not equal to • & bitwise AND • ^ bitwise XOR • | bitwise OR • && logical AND • || logical OR • ?: conditional

22.3 FUNCTIONS

Below is a list of functions available for use as needed.

Function	Definition	Usage
arrayValue	Uses an index expression to lookup the value in an array Constant. If the index expression is not a whole number, the value will be interpolated. Out of range indexes will return the nearest railed out value.	arrayValue(array.[aConstantArray], [indexExpression]) Note: the array constant name must be prefaced by “array.” for the pre-parser to properly map the constant
Smoothing*	Smooths a field by averaging over the smoothingFactor number of records.	smoothBasic(field, smoothingFactor)
sine	Standard Sine of a value.	sin(val)
cosine	Standard Cosine of a value.	cos(val)
arcsine	Standard arcsine of a value.	asin(val)
arccosine	Standard arccosine of a value.	acos(val)
tangent	Standard Tangent of a value.	tan(val)
arc tangent	Standard Arc Tangent of a value.	atan(val)
square root	Standard Square Root, same as pow(val, 0.5) of a value.	sqrt(val)
absolute	Changes any negative values to the same magnitude in the positive direction.	abs(val)
log	Natural log of a value.	log(val)
log base 10	Base 10 log of a value of a value.	log10(val)
reciprocal	The reciprocal of a value, or 1/val	recip(val)
exponent	Exponent of a value.	pow(val, exponent)
round	Rounds the value of a value to the nearest integer value.	round(val)
floor	Returns the largest (closest to positive infinity) integer value that is greater than or equal to the argument.	floor(val)
ceiling	Returns the smallest (closest to negative infinity) integer value that is greater than or equal to the argument.	ceil(val)
exponent	Returns Euler's number e raised to the power of a double value.	exp(val)
isNaN	Checks the output of an expression to result in a invalid number	isNaN(val)
smoothFiltered	Smooth a field using a matrix filter for less lag	smoothFiltered(val)

accelHp	Calculates HP based on Acceleration rate.	accelHp(velocity (MPH), deltaVelocity(MPH), deltaTime(s), weight(lb))
Aero Drag	Calculates the Aerodynamic Drag	aerodynamicDragHp(velocity (m/s), airDensity (kg/m^-3), dragCoefficent, frontalArea (m^2))
Rolling Drag	Calculates estimated rolling resistence	rollingDragHp(speed (MPH), tirePressure (psi), weight (lbs))
Last Value	Returns the last record value for the specified field or expression	lastValue(anyField)
Historical Value	Returns the value for the specified field or expression from n records back	historicalValue(anyField, n)
Min	Returns the minimum value resulting from 1-n expressions or variables	min(exp, exp, ...)
Max	Returns the maximum value resulting from 1-n expressions or variables	max(exp, exp, ...)
Max Value	Returns the maximum historical value for the specified field or expression	maxValue(anyField)
Min Value	Returns the minimum historical value for the specified field or expression	minValue(anyField)
selectExpression	Uses the 1st expression as the index of the expression to use. This can have 1 to n expressions listed starting with index 0 as the 2nd parameter after the index expression. The Index Expression is rounded to the nearest integer value.	selectExpression(indexExpression, expression0, expression1, …, expressionN)
table	Perform a table lookup using an inc file	table(expression, 'fileName.inc')
tableLookup	Perform a table lookup on 1D or 2D arrays based on specified OutputChannel(s) and 1D array lookup references. It will provide the interpolated value for the relative position of the lookup.	tableLookup([array.valueArrayName], [array.lookupArrayName], [lookupChannelName]) tableLookup([array.zParamName], [array.xParamName], [array.yParamName], [xChannelName], [yChannelName]) Note: the array constant name must be prefaced by “array.”

Accumulate*	Accumulate and sum the expression This allows totaling distance, mileage, fuel consumption, etc.	accumulate(expression)
Persistent Accumulate*	Same as accumulate, except the accumulated value is persisted to the next session. Allows for an Odometer.	persistentAccumulate(expression)
Get Main CAN_ID	Returns the CAN ID of the primary controller in a project.	getWorkingLocalCanId()
getChannelValueByOffset	Will retrieve the user value of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelValueByOffset(offset)
getChannelScaleByOffset	Will retrieve the scale of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelScaleByOffset(offset)
getChannelTranslateByOffset	Will retrieve the translate of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelTranslateByOffset(offset)
getChannelDigitsByOffset	Will retrieve the display digits of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelDigitsByOffset(offset)
getChannelMinByOffset	Will retrieve the min of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelMinByOffset(offset)
getChannelMaxByOffset	Will retrieve the maximum of a Controller OutputChannel that has an offset matching the parameter. The offset can be an expression that resolves to an integer	getChannelMaxByOffset(offset)
getLogTime	Returns the length in seconds of the currently being captured data log. If not logging returns -1	getLogTime()
isOnline	Reports online state. Will return 1 if connected to the controller, 0 if offline.	isOnline()
isAdvancedMathAvaliable	Some advanced functions are unavailable in some applications or additions. This will report 1 if they are available.	isAdvancedMathAvaliable()

23 STRING FUNCTIONS

23.1 OVERVIEW

String functions are used to create String values that can be changed at runtime based on any set of conditions. They are commonly used for Data Log Headers, Titles, Labels and Units. The StringFunctions are supported by Constants, OutputChannels, Gauge Templates and Dialog Components.

23.1.1 Available Functions

23.1.1.1 bitStringValue([bitConstantName], [expression])

At runtime, this allows a segment of a string to be replaced with an option defined for a bit type Constant. The expression should resolve to the desired index. This requires a bit Constant and an expression that resolves to the proper index.

23.1.1.2 Example

Here we will create an indicator that will report a trouble based on a trouble code.

```
[PcVariables]
    troubleList  = bits,   U08,   [0:2], "Ok", "O2 Sensor", "CLT Sensor", "IAT Sensor"

[FrontPage]
 indicator = { malFunc }, "No Codes", {Error: bitStringValue(troubleList, malFunc)},   green, black, red, black
```

The above relies on an OutputChannel that is 0 if there is no code, under sensor error, it will be a value matching the index of the failed sensor. The result will be an indicator that will follow this matrix:

malFunc value	Indicator state
0	Green Indicator with text: “No Codes”
1	Red indicator with black text: “Error: O2 Sensor”
2	Red indicator with black text: “Error: CLT Sensor”
3	Red indicator with black text: “Error: IAT Sensor”

23.1.2 stringValue([StringConstantName])

The stringValue() function is to include the value of a string Constant in a String. This is useful in situations where you allow a user to name something such as a generic input using a string Constant or PcVariable. The User name can then be referenced for data log field names and gauge titles.

The only parameter is the string Constant or PcVariable you wish to access.

23.1.2.1 Example Usages:

```
[GaugeConfigurations]
   sensor01Gauge= sensor01,{ stringValue(sensor01Alias) }, "",-10, 10, -10, -10, 10, 10, 1, 1

[Datalog]
   entry = sensor01, { stringValue(sensor01Alias) }, float, "%.1f"
```

sensor01Alias is assumed to be a string Constant or PcVariable with some value.

The application is responsible for resolving these string functions dynamically at runtime.

23.1.3 $getProjectsDirPath()

Provides access to the applications projects folder, This can be embedded in a string. At runtime this will be replaced with the absolute path to the applications projects directory. Can be useful if you want one common directory for all projects such as to access help files in a single folder, not in the specific project folder.

23.1.3.1 Example Usage:

```
[UiDialogs]
    dialog = myDialog, "", yAxis
         topicHelp = "file://$getProjectsDirPath()/docs/Reference-1.4.pdf#deeplink"
        field = "Some field", someConstant
```

In this case $getProjectsDirPath() is used to access a common docs folder in the application projects folder.

23.1.4 $getWorkingDirPath()

Similar to $getProjectsDirPath(), but $getWorkingDirPath() provides access to the specific project that the ECU Definition is a part of.

24 [FTPBROWSER]

A file browser to list, download and delete files from the controller is supported. This component uses an FTP protocol, so is only available when connected via IP and TCP is available.

The FTPBrowser section allows configuration for FTP Browsers with 1 or more end points.

24.1 DEFINING AN FTP BROWSER:

ftpBrowser = referenceName, "Title", { enableCondition }

enableCondition is Optional.

24.1.1 Attributes

host (optional) – if not used, the connected controller is the expected target.
port (optional) – if not defined port 22021 is used.
user (optional) – if not defined anonymous is used. Ensure proper permissions
password(optional) – if not defined sd@efianalytics.com is used as an anonymous pw.
browseEnabled (optional) – expression to enable listing files.

```
browseEnable = "Disabled user message", { expressionforEnabled }
```

readWriteEnable (optional) – expression to enable reading and deleting files.

```
readWriteEnable = "Disabled user message", { expressionforEnabled }
```

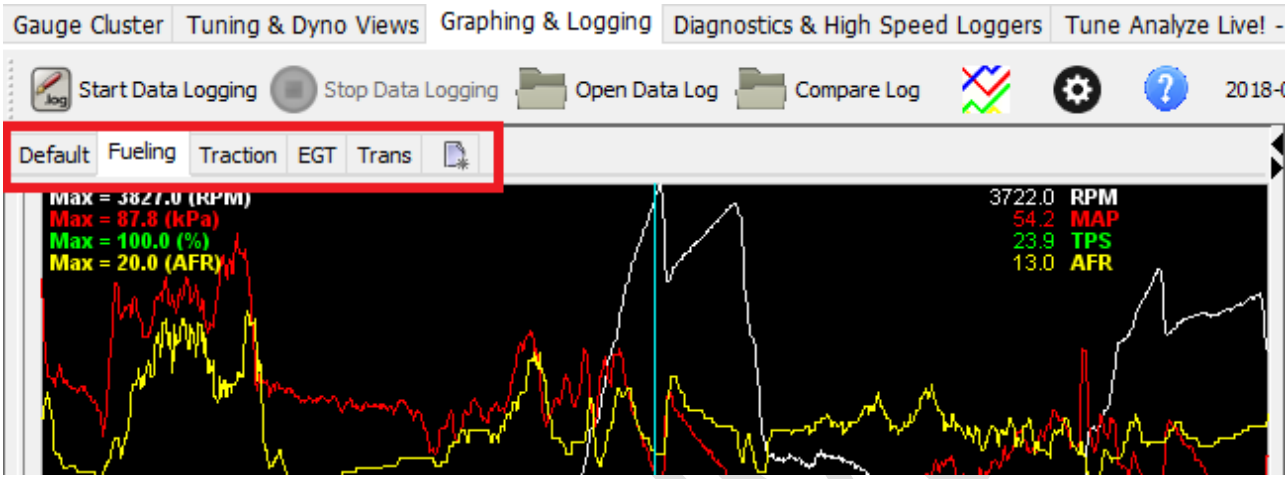
if expressions are not present, they are assumed to always be true.

```
[FTPBrowser]
```

```
;ftpBrowser = referenceName, "Title", enableCondition
ftpBrowser = sdCardBrowser, "SD Card 1 File Browser"
    ; All below are optional. if any of these settings is not present, the default value will be used.
    host = connectedController ; (default value = connectedController)
    port =  22021; (default = 22021)
    user = anonymous ; (default: anonymous)
    password = sd@efianalytics.com ; (default = sd@efianalytics.com)
    ; browseEnable = "Disabled user message", { expressionforEnabled }
    browseEnable = "Please Turn off the engine to view SD files", { isNaN(FILT_RPM) || FILT_RPM < 50 }
    ; readWriteEnable = "Disabled user message", { expressionforEnabled }
    readWriteEnable = "Please Turn off the engine to download or delete files", { isNaN(RPM) || RPM < 50 }
```

25 [DATALOGVIEWS]

The [DatalogViews] allows for the definition of predefined LogViews / quick Views available in editions that have an integrated log viewer. iniSpecVersion 3.60 is required.




```
[DatalogViews]

; 1st one with no name will be the Default traces for the front tab
; each trace is defined as graph[graphNumber].[traceOnThatGraph]
; thus graph0.0 will be the 1st trace on the top graph, graph0.1
; will be the 2nd trace on the top graph.
; graph1.0 will be the 1st trace on the 2nd graph, etc...
graph0.0="RPM"
graph0.1="MAP"
graph0.2="TPS"
graph0.3="AFR"
graph1.0="CLT"
graph1.1="MAT"
graph1.2="Batt V"

; Now create Additional names tabs. The logViewName tag indicates a new Tab,
; The string provided will be the name of the tab displayed to the user.
logViewName = "Fueling"
graph0.0="RPM"
graph0.1="MAP"
graph0.2="TPS"
graph0.3="PW"
graph1.0="CLT"
graph1.1="MAT"
graph1.2="Batt V"
graph2.0="AFR"
graph2.1="AFR 1 Error"
graph2.2="AFR Target"
graph2.3="EGO cor1"
graph3.0="Duty Cycle1"
graph3.1="VE1"
graph3.2="Fuel: Accel PW"
graph3.3="Fuel: Warmup cor"
```