M1 DEVELOPMENT

local UseGPS = Delay.Falling(speed <= 0.0 and Root.GPS.Speed > 2.78 /* 10kph */, Diagnostic Delay);

Value = Root.GPS.Speed; State = State.GPS; } else if (fault or Vehicle.Speed.State eq Vehicle.Speed.State.Estimated)

Value = 0.0; /* hold previous State */

AUTHORISED M1 PARTNER PACKAGE

MANUAL

MoTeC

Printed documents are not controlled.

While every effort has been taken to ensure correctness, MoTeC, its employees, contractors and authorised representatives and agents take no responsibility, make no express or implied guarantees, representations or warranties to any third party in relation to consequences arising from any inaccuracies or omissions in this document.

This is a proprietary document and the property of MoTeC Pty. Ltd. No part of this publication may be reproduced in any retrieval system, or transmitted in any form or by any means, whether electronic, graphic, mechanical, photocopy, recorded or otherwise, without the prior express written permission of MoTeC.

MoTeC reserves the right to make changes to this document without notice.

Copyright 2014

All rights reserved.

Table of Contents

Introduction

Thank you for choosing the MoTeC M1, a new generation of Engine Control Units (ECUs).

In this manual, **M1 Build** refers to the M1 Build software and **M1 Tune** to the M1 Tune software.

The purpose of this manual is to assist application developers to produce applications for the MoTeC M1 series of ECUs. It contains descriptions of the functional elements that make up an M1 Build Project. This manual, together with the integrated help in M1 Build, can be used as a programmer's reference.

☛ *A high level of assumed knowledge is required to use the M1 Build and M1 Tune software. This assumed knowledge is not elaborated upon, and is outside the scope of this manual.*

Terminology

M1 introduces unique terms and at times uses common terms to represent something specific to the M1 environment. Where common terms are used in this fashion they are specially formatted to identify this use.

See the Glossary for a list of terms, their definitions and where applicable, the formatting used.

Assumed Knowledge

The knowledge recommended for users of this manual is listed below. This manual does not attempt to provide any of this information.

Mechanical:

- 4 stroke engine cycle
- Camshaft actuators
- Throttle servo
- Turbocharging
- Supercharging
- Ignition systems
- Fuel systems.

Calibration:

- Engine mapping (fuel and ignition)
- Boost control
- Camshaft timing
- Throttle servo control
- Control systems (e.g. PID, Traction).

Programming:

- Some experience with structured programming
- Experience with C, C++, C# or Java may be useful.

The MoTeC M1 Framework

The MoTeC M1 ECU is a programmable electronic control unit. This means that the behaviour of the M1 ECU can be defined in a similar fashion to that of a traditional software application. M1 applications are constructed using M1 Build.

The M1 framework has the following high level structure:

- M1 Build and M1 Tune
- M1 ECU software stack
  - M1 System
  - SDK
  - M1 Project (the compiled output of the Project is the Package)
  - M1 Package
    - Firmware
    - Meta-data
    - Stored calibration
    - Logging configuration
    - Security configuration
- Hardware.

M1 Build and M1 Tune Overview

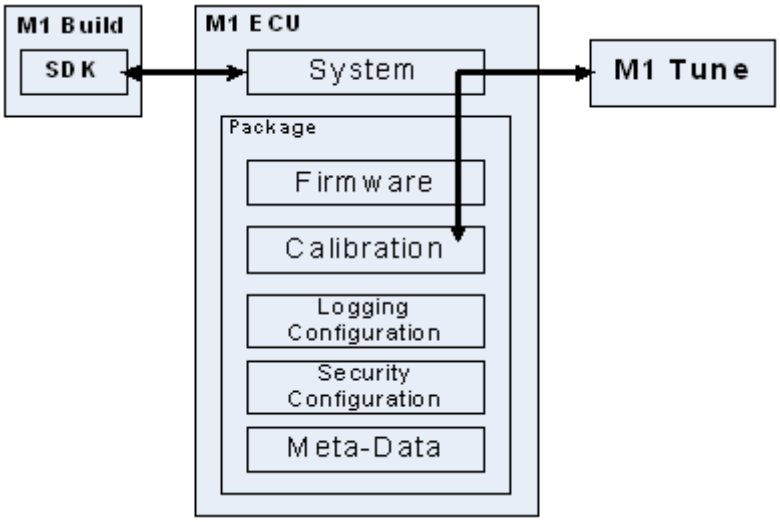
M1 Build is an integrated development environment tailored specifically for use with the M1 ECU. At the highest level it contains a functional description of the application, a code editor and a compiler. An M1 Build Project is compiled into a machine code firmware for the target M1 ECU hardware. During the compilation process, meta-data containing information about the contents of the firmware is generated. This meta-data describes memory layout and security details which are required for calibration. A compiled Project containing firmware and meta-data is called a Package. Packages can be calibrated and sent to an M1 ECU using M1 Tune.

M1 Tune is a calibration environment designed to calibrate Packages. As the Package contains meta-data describing its memory layout, M1 Tune is able to calibrate any Package built using a compatible version of M1 Build.

M1 ECU Software Stack Overview

The M1 ECU software stack consists of two major components: the M1 System and the M1 Package.

M1 ECU Software Stack



M1 System

The M1 System defines the operating environment for the Package. It consists of a kernel, drivers for the input/output hardware, file systems and management tasks. Memory protection is employed to prevent the firmware and other tasks from performing unauthorised memory accesses. The M1 System manages the operation of the firmware and provides additional services as detailed below.

The management task provides a communications interface which M1 Tune uses to communicate with the firmware and the rest of the system. Before any calibration or other adjustments to the ECU can occur, security checks are performed. The nature of these checks is dependent on settings in the Package.

The final responsibility of the M1 System is to perform data logging as requested by the M1 Package.

Software Development Kit (SDK)

The SDK contains definitions of the system functions and input/output methods available to M1 Build for use in a Project.

The SDK and System are published as a matched set. The SDK selected for the Project defines the System that will be required to run the compiled Package in the M1 ECU. M1 Tune ensures the M1 ECU is running the correct System when sending an M1 Package and updates the System when required.

M1 Project

A Project is the complete source-level definition of an M1 ECU application.

It contains a collection of all objects, data types, security groupings and the logic for the M1 ECU firmware.

Projects are written using the capabilities exposed by the SDK.

Package

A Package is the output of an M1 Build Project compilation. Packages contain firmware, meta-data, non-volatile calibration data, data logging descriptions and security definitions. A Package represents the complete state of an M1 ECU. Packages can be opened and calibrated using M1 Tune and contain all information required for M1 Tune to display Package data. Packages can be sent to and retrieved from an M1 ECU.

Firmware

The firmware is the machine code generated by the M1 Build Project compilation. This compilation is one-way and cannot be reversed.

The firmware operates on two distinct types of information; calibration and data. In general:

- Calibration consists of information required by the firmware, for example, fuel maps and sensor calibrations.
  - Calibration is modified by M1 Tune. It cannot be modified by the firmware.
- Data consists of information generated by the firmware, for example, current fuel volume and calibrated sensor value.
  - Data is modified by the firmware. It is not modified by M1 Tune.

Meta-data

The Package meta-data contains information required to calibrate the Package. This includes the location, size, units and name of all data and calibration objects defined in the Project and includes:

- Definitions of data generated by the firmware. Data is read only to M1 Tune.
- Definitions of calibration used by the firmware. Calibration is read only to the firmware.
- Definitions of security groups.
- M1 Tune Worksheets and Workbooks.

Stored Calibration

The Package calibration contains a non-volatile copy of the calibration values used by the firmware.

When the firmware executes, a volatile copy of the calibration is created in the M1 ECU's RAM. The volatile copy is used during firmware operation and calibration. The non-volatile calibration is updated at the request of M1 Tune; this process is initiated in response to a user action. See the M1 Tune manual for more information.

Logging Configuration

The Package logging configuration defines the data to be logged to the M1 ECU's flash memory.

There are eight (8) separate log systems in the M1 ECU, and each can have a different logging configuration. Each logging configuration contains a list of data items and the rates at which the data items are to be logged.

☛ *In the current version, calibration values cannot be logged.*

Security Configuration

The Package security configuration defines users and access rights to the Package. The security configuration is limited by the security groups defined in the Project.

M1 Hardware Overview

Hardware variants, specifications and features are included in the separate M1 ECU Hardware manual.

Classes and Methods

☛ *Classes described in this manual are in accordance with version 1.4.0.0092 of M1 Build.*

Classes

Classes are divided into different class categories. Built-in and event classes are the lowest level building blocks. They can be used directly or to build other classes.

Hardware classes are used to interact with physical hardware in the M1 ECU.

Higher level classes are constructs built up of other classes. In general such a class comprises several elements. These typically include channels, tables, value inputs, IO resource inputs and methods.

Built-in and event classes are described in the following sections of this manual. Hardware and Higher level classes are described in the integrated M1 Build Help.

Any instance of a class which is included in a Project is called an object. Each object in a Project must have a unique name.

Methods

A method is a function associated with an object. Methods have the special property that at the point of execution, they have access to the data associated with an object. All methods in the M1 System are statically bound.

Methods

Commonly used methods on M1 objects are:

`AsInteger()`

Converts an enumerator to its integer representation, returning an integer value.

`AsString()`

Converts an enumerator to its string representation, returning a string.

`Set(value v)`

Sets the object bound to the method to the value v.

`Validate(value v)`

Checks whether v is a valid value for the object bound to the method, returning a boolean result.

Methods are accessed in code using a dot '.' after the object's name.

EXAMPLE

The following statement:

```
Engine.State.AsInteger()
```

executes the method:

```
AsInteger()
```

on the object:

```
Engine.State
```

Built-in Classes

Built-in classes can be grouped into three categories:

- Calibration
- Data
- Software.

Calibration Classes

Calibration classes can be viewed and changed by M1 Tune. The M1 ECU firmware cannot modify the value of a calibration class.

Parameter

A parameter represents a single value which can be viewed and changed by M1 Tune. The units and allowable range for the parameter can be configured.

EXAMPLE:

Name	Base Unit	Display Unit
Fuel Overall Trim Fuel.Volume.Trim Overall 15.0 %Trim	ratio	%trim
Traction Control Activation Speed	m/s	km/h

Tables

A table is a matrix of values which can be configured with 1, 2 or 3 axes.

Each configured axis is allocated an object which is used to perform an interpolated look up on the matrix. The table look up is performed by the scheduled Update() method, and the look up result is stored in a value channel. In addition, in a code, a table value can be read using the Lookup()-method.

The units and allowable range for the matrix values can be configured.

Input values may be either a floating point value or an enumerated value. Integer values are not allowed.

The lookup result is a floating point value.

If the current value of an axis channel is between two axis values, the resulting table values are always interpolated linearly using the lookup values of the two axis values adjacent to the current channel value. In M1 Build, tables can also be configured to extrapolate below an axis minimum, above an axis maximum or in both directions. This extrapolation is also done linearly, using the table values of the first two axis values or the last two axis values, respectively.

The axis values and the table body data values are calibrated in M1 Tune. Each axis, if enabled, must be calibrated with values in ascending order. Independent from the axis values, each axis site can be addressed by firmware methods by its index, which is an ascending integer number starting with 0 for the first site.

EXAMPLE:

Name	Axes	Base Unit	Display Unit
Fuel Volume	Engine Efficiency, Engine Speed	m^3	μL
Ignition Timing	Engine Load, Engine Speed	°	°BTDC

Fuel.Volume.Main [µl]

Engine.Efficiency [%] \ Engine.Speed [rpm]	0.0	800.0	1000.0	1500.0	2000.0	2500.0	3000.0
240.0	130.0	130.0	130.0	130.0	130.0	129.0	129.0
220.0	130.0	130.0	130.0	130.0	130.0	129.0	129.0
200.0	130.0	130.0	130.0	130.0	130.0	116.3	131.7
180.0	130.0	130.0	130.0	130.0	130.0	116.3	129.8
160.0	130.0	130.0	130.0	130.0	130.0	107.1	117.3
140.0	128.0	128.0	128.0	128.0	128.2	94.4	86.5
120.0	60.0	60.0	60.0	50.3	75.4	83.4	76.2
100.0	50.0	48.0	48.0	74.9	72.9	61.6	51.8
80.0	50.0	45.0	45.0	43.3	37.4	44.2	43.3
60.0	50.0	35.0	28.4	28.3	27.1	30.8	32.8
40.0	40.0	25.0	11.6	10.4	16.4	16.5	18.7
20.0	30.0	25.0	11.0	11.3	10.0	10.0	10.0
0.0	20.0	28.0	13.5	11.3	10.0	10.0	10.0

Firmware Methods for Tuning Tables

`Constrain()`

Constrains the supplied value to the object's limits.

- Value > 'Validation Maximum' returns 'Validation Maximum'
- Value < 'Validation' Minimum returns 'Validation Minimum'
- Otherwise returns Value

The value of the object is not changed

`Update()`

This method performs the interpolated lookup on the table which updates the table value. Must be scheduled against a periodic event.

`Init()`

Initialises the table. Automatically scheduled against the startup event.

`Validate(value v)`

Checks whether v is within the validation range for the table. Returns a boolean value.

```
XAxisMaximum()
YAxisMaximum()
ZAxisMaximum()
```

Returns the maximum value on the respective axis, or 0 if the axis is disabled. Only available if the data type of the axis object is floating point.

```
XAxisMinimum()
YAxisMinimum()
ZAxisMinimum()
```

Returns the minimum value on the respective axis, or 0 if the axis is disabled. Only available if the data type of the axis object is floating point.

```
Lookup (x axis value [, y axis value] [, z axis value])
```

Lookup the table with the provided axis values.

```
GetUnscheduled()
```

Get the channel value but do not use this in scheduling analysis.

This method should only be used to prevent a circular scheduling dependency. For example:

- Scheduled Function 1 reads Channel State and writes Diagnostic
- Scheduled Function 2 reads Channel Diagnostic and writes State

This is a circular reference as the scheduler does not know if Scheduled Function 1 or Scheduled Function 2 should execute first. If Scheduled Function 2 is amended to use Diagnostic.GetUnscheduled() the scheduler will exclude this channel from the dependency analysis and can then detect that Scheduled Function 2 should execute first, so Scheduled Function 1 can use the latest value of State.

```
XAxisGet()
YAxisGet()
ZAxisGet()
```

Gets the axis site value at the specified index. If index is not provided, the index calculated by Update() is used.

```
XAxisIndex()
YAxisIndex()
ZAxisIndex()
```

Determines the index of the lowest site used when interpolating with the supplied value. If value is not provided, use the same site as the last execution of Update().

```
XAxisIndexFirst()
YAxisIndexFirst()
ZAxisIndexFirst()
```

Determines the index of the first site of the axis. This value is always 0.

```
XAxisIndexLast()
YAxisIndexLast()
ZAxisIndexLast()
```

Determines the index of the last enabled site of the axis.

```
XAxisPosition()
YAxisPosition()
ZAxisPosition()
```

Determines the position on the axis relative to the lowest site used when interpolating with the supplied axis channel value. If no value is provided, the same site as the last execution of Update() is used. The result is <0 if the supplied axis channel value is below the minimum value of the axis (the value on the first axis site), or >1 if the supplied axis channel value is above the maximum value of the axis (the value on the last axis site). The result is between 0 and 1 if the supplied axis channel value is between 2 site values, for example 0.5 when midway between 2 site values.

`Get()`

Gets the body value at the provided index values. If index is not provided, use the index of the lowest body site calculated by Update().

`Hash()`

Detects if the calibrated components of the table (axis and/or body) have been edited.

Data Classes

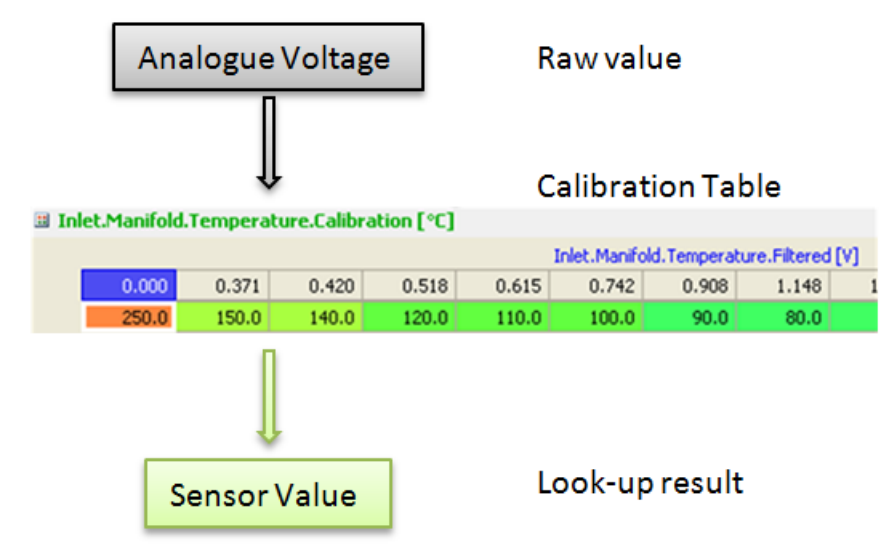
Data classes can be viewed but not changed by M1 Tune. The M1 ECU firmware writes the value of data classes. Data Logging can record these values in the M1 ECU internal memory.

Channel

A channel represents a single value written by the M1 ECU firmware. The units and allowable range for a channel can be configured.

EXAMPLE

A channel is used to represent a sensor value as follows:



Methods for Channel

`Set(value v)`

Sets the value of the channel to v. Clamps v to within the minimum and maximum validation range.

`Validate(value v)`

Checks whether v is within the validation range for the channel. Returns a boolean value.

`Constrain(value v)`

Constrains the supplied value to the object's limits.

`AsInteger()`

For enumerated channels only.

Returns the integer representation of the current enumerated value of the channel.

`AsString()`

For enumerated channels only.

Returns the string representation of the current enumerated value of the channel.

`GetUnscheduled()`

Get the channel value but do not use this in scheduling analysis.

Learning Tables

Learning Tables equal Tuning Tables with the only exception that the body values are not calibrated by the user in M1 Tune but set by the firmware using associated methods.

Methods for Learning Tables

Learning tables have all methods of Tuning Tables associated, plus the following.

`Set()`

Sets the lowest table body site used when Update() last executed to the supplied value.

If the optional axis index values are provided, the indexed table body site is set to value.

`Preset()`

Presets all table body sites to a specified value.

`Adjust`

Adjust the table body sites towards a provided target value. The number of affected body sites around the given index can be defined. Affected body sites will be adjusted linearly relative to their distance from the given index.

Also the change ratio can be provided.

Software Classes

Software classes are used in the construction of an M1 application and are not directly visible from M1 Tune. They comprise the following elements:

- Constant
- Function
- Scheduled Function
- Tuning Method
- Group
- Timer

EXAMPLE

Software Classes

Name	Class	Quantity	Data Type
Root	Group		
Events	Group		
On 100Hz	Event		
On 10Hz	Event		
Constants	Group		
Maximum Speed	Constant	Unitless	Floating Point
Gears	Constant	Unitless	Floating Point
Calculation	Scheduled Function		
Update	Scheduled Function		

Constant

A constant is a single value that cannot be viewed, logged or changed from within code or M1 Tune. A constant is used for a number of reasons:

- Clarity
  A constant can be used to make a calculation easier to read or understand, such as a reference to the number of cylinders.
- Unit Conversions
  To ensure that a constant value is entered correctly for the system base unit.
- Package Control
  A constant cannot be modified once the Project has been compiled. This can be used to stop the end user making undesired modifications.

Function, Scheduled Function, Tuning Method

These classes are containers for M1 Programming Language (M1PL) code.

This code can be used for logic and calculations in a Project. The syntax of an M1PL code is similar to C.

EXAMPLE

Function, Scheduled Function and Tuning Method

```
1 when (Type)
2 {
3     is (Sawtooth)
4     {
5         Signal = Counter / 360.0 ;
6     }
7     is (Square)
8     {
9         if (Counter > 180)
10        {
11            Signal = 1.0;
12        }
13        else
14        {
15            Signal = 0.0;
16        }
17    }
18    is (Sine)
19    {
20        Signal = ((Calculate.FastSin(Counter) + 1.0) * 0.5);
21    }
22 }
23
```

A more detailed description of M1PL, its use, and syntax can be found in The M1 Programming Language topics.

Functions are executed when they are called, whereas scheduled functions are associated with an event.

Functions can optionally have input arguments and a return value which are defined by the user (see the 'Properties' topic in the M1 Build User Manual for more details). These optional values can be accessed inside the function using the 'In' and the 'Out' keywords.

Tuning methods provide firmware interaction with M1 Tune to allow table or parameter values to be updated. The Tuning Method class is only executed upon demand. When the 'Q' key is pressed in M1 Tune the current input value is sampled and the method is executed.

Calibration functions can assign a value to their parent Table or Parameter using the syntax:

```
this = [value];
```

When assigning to a Parameter the value is assigned as the new value for the Parameter.

When assigning to a Table the value is stored in the current site on the Table.

The method may use complex operations to manipulate the value before it is stored in the relevant table or parameter. Note that there are library functions designated for use only in calibration functions, see Library Functions for Calibration Methods.

Methods for Tuning Methods:

Apart from the methods associated to Tuning Tables that have been listed in chapter 'Calibration classes', tuning methods provide the following additional associated methods

```
XAxisIndexFirmwareFirst()
YAxisIndexFirmwareFirst()
ZAxisIndexFirmwareFirst()
```

Determines the first available index of an interpolated axis (always 0).

```
XAxisIndexFirmwareLast()
YAxisIndexFirmwareLast()
ZAxisIndexFirmwareLast()
```

Determines the last available index (that is the Maximum Sites defined in M1 Build - 1).

```
XAxisValidate()
YAxisValidate()
XAxisValidate()
```

Tests the supplied value against the axis' validation limits. The value of the object is not changed.

```
XAxisConstrain()
YAxisConstrain()
XAxisConstrain()
```

Constrains the supplied value to the axis' validation limits.

- Value > 'Validation Maximum' returns 'Validation Maximum'
- Value < 'Validation Minimum' return 'Validation Minimum'
- Otherwise returns Value

The value of the object is not changed.

```
XAxisSet()
YAxisSet()
XAxisSet()
```

Sets the axis site at the provided index to the supplied value. This will fail if value is not valid.

```
Mark(Boolean)
```

Sets (true) or Clears (false) the mark on the current table site or parameter which indicates a value change in M1 Tune.

```
Validate(value v)
```

Checks whether v is within the validation range for the object. Returns a boolean value.

```
Constrain(value v)
```

Constrains the supplied value to the object's limits.

If the supplied value is outside the range of the object's Validation Minimum/Maximum the value assigned will be clamped to the nearest Minimum/Maximum value.

EXAMPLE

In the code of a calibration method:

```
This = This.Constrain(new);
This.Mark(false);
```

will set the current parameter or table site to the clamped value of a local variable new and clear the mark on that parameter or table site.

Online and Offline Calibration

The M1 Tune environment implements calibration scripts differently when the M1 ECU is online or offline:

Online M1 ECU:

- All channels are evaluated and if successful the results are written into the respective table or parameter that is addressed by the calibration function. User prompts may be included when some conditions are not met.

EXAMPLE

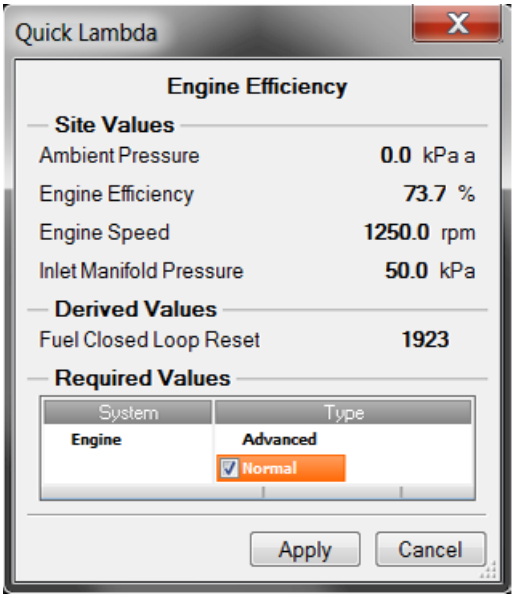
GPR Fuel Efficiency Value 'Quick Lambda' Tuning Method (part):

```
if (Math.IsNaN(Exhaust.Lambda.Normalised))
{
UI.PromptOK(
"Quick Lambda",
"No lambda measurement available."
);
}
```

In this example a user prompt advises the tuner that no Lambda measurement is possible.

Offline M1 ECU:

Many scripts may still execute by offering a user prompt, allowing manual entry of the required channel values.



In this case the M1 Tune user may enter values for Ambient Pressure etc. and, if the expression evaluates correctly, the result will be written into the Engine Efficiency table.

Overly complex calibration scripts may baffle or deter the Tune user when prompted for manual entry, so it is best to keep channel entry requirements to a minimum for this circumstance.

Group

The Group class is a container of objects. It has no inherent characteristics. Groups are used to structure a Project in a logical fashion. Typically, it is used to create functional blocks such as 'engine', 'airbox', or 'transmission', under which other objects are collected.

Throughout the M1 System a "." (period) is used as a group separator. For example "Fuel.Cylinder 1.Trim" is an object called "Trim" inside a group called "Cylinder 1" inside a group called "Fuel".

Although the application developer has full freedom on how to group objects in a Project, well thought out and consistent grouping will make the Package easier to navigate in M1 Tune.

EXAMPLE

The following is an example of a group hierarchy:

```
Fuel.Aim
Fuel.Pump
Fuel.Level
Fuel.Used
Fuel.Volume.Main
Fuel.Volume.Compensation.Inlet Manifold Pressure
Fuel.Volume.Trim.Overall
Fuel.Cylinder 1.Pulse Width
Fuel.Cylinder 1.Trim
```

In conjunction with M1 Tune, object hiding rules are applied related to groups. If a group contains a parameter or an IO Resource Parameter that assumes a value of 'Not in Use', all other objects within this group will not be displayed in M1 Tune. This can be used to hide objects of control systems or sensors that are not used in the current application, thus providing the user only with objects in use.

Timer

This class provides a countdown timer which may be used within code to effect time-outs and other functions. The accuracy of the timer is high, but the effectiveness of its application is dependent upon the execution rate of the code from which it is monitored.

Methods for Timer

`Start(value v)`

Sets the timeout time for the timer. v is a time in seconds.

`Remaining()`

Returns the remaining time before the timer expires. This function returns a floating point value which is less than or equal to zero when the timer expires.

IO Resources

An IO Resource represents a connection between the firmware and an external interface.

These resources can be physical pins on the ECU connector, internal measurements or software services provided by the system.

For example, an “Absolute Voltage” input can be configured to use the resource “Analogue Temperature Input 1” or “Battery Positive”.

IO Resource Constant

An IO Resource Constant has the same function as a Constant but is used to represent an IO Resource.

IO Resource Parameter

An IO Resource Parameter has the same function as a Parameter but is used to represent an IO Resource.

Event

Events are triggers for performing an action in an M1 firmware. When an event triggers, the firmware can:

- Run scheduled functions
- Run scheduled methods
- Update analogue values
- Set output pulse widths, etc.

The application developer is able to select an event for each object based on the events configured in the Project.

If a scheduled function is to be executed at 50Hz, then a 50Hz event must be added to the Project. Normally all Projects require a Startup event for initialisation, and individual events to supply the required calculation rates. Typically all events are contained within a group called Events.

The order of execution of objects scheduled against the same event is automatically determined by M1 Build.

One special case is Dual Rate Event which allows slower rates under normal conditions, and higher rates under certain defined conditions. Events are:

- Dual Rate Event
- Event (On 1000Hz)
- Event (On 100Hz)
- Event (On 10Hz)
- Event (On 1Hz)
- Event (On 200Hz)
- Event (On 20Hz)
- Event (On 2Hz)
- Event (On 500Hz)
- Event (On 50Hz)
- Event (On 5Hz)
- Event (On Shutdown)
- Event (On Startup).

Data Types

All data types in the M1 ECU are 32 bits in width.

Application developers can define values to be one of the following data types.

- Floating Point
- Enumeration
- Integer
- Unsigned Integer
- Boolean (local variables only)
- String (local variables only).

Floating Point

Floating point represents real numbers in a way that can support a wide range of values. Numbers are represented by a fixed number of significant digits and scaled using an exponent.

```
Type: IEEE 754-2008 binary32 floating point number.
Significand Precision: 23-bits
Exponent Width: 8-bits
Sign Bit: 1-bit
```

Enumeration

Enumerations are used where it makes more sense to use a textual description rather than a numeric value.

EXAMPLE

"Engine.State" could be represented by the enumeration:

-1 = Error

0 = Stopped (default)

1 = Cranking

2 = Idle

3 = Running

4 = Overrun

In the above example, the string "Idle" will be displayed in M1 Tune when the engine is idling, rather than the numeric value 2. This makes a Package easier to manage and understand.

Enumerations can be user defined or built-in, but they must be configured in M1 Build by the application developer. The numeric representation is automatically managed by M1 Build but a default value must be defined. See the M1 Build User Manual for details.

Enumerators can be accessed in code using a dot '.' after the object's name or the enumeration name.

EXAMPLE

Accessing an enumeration using dot (.)

```
if (Engine.State eq Engine.State.Idle)
{
...
}
```

Enumerated objects provide the following methods:

`AsString()`

Converts an enumerated object to its string representation.

`AsInteger()`

Converts an enumerated object to its integer representation.

EXAMPLE

Equivalent to the previous example is the following expression:

```
if (Engine.State.AsInteger() eq 2)
{
...
}
```

Enumerators can be assigned an indicator. The indicator is used by M1 Tune to determine the importance of each enumerator. If a channel value matches an enumerator that is defined to be 'Information', the channel will be shown in the M1 Tune status list. If the enumerator is defined to be 'Warning' or 'Fault', the channel will contribute to the warning or fault count that is presented to the user in M1 Tune. This provides a powerful mechanism to draw the user's attention to abnormal system behaviours.

Integer

An integer is a whole number that is positive, negative or zero.

```
Size: 32-bits
Minimum value: -2,147,483,648
Maximum value: 2,147,483,647.
```

Integer values are most commonly used in communications code.

Unsigned Integer

An unsigned integer is a non-negative whole number.

```
Size: 32 bits
Minimum value: 0
Maximum value: 4,294,967,295.
```

Unsigned integer values are most commonly used in communications code.

Fixed Point 7dps

This Fixed Point data type is an integer scaled by 1e-7 to represent a positive or negative number with fixed precision.

```
Size: 32-bits
Minimum value: -214.7483648
Maximum value: 214.7483647
```

Fixed point data types are most commonly used in GPS related code.

☛ *Fixed point data types support a limited subset of arithmetic operations.*

Boolean

Boolean data types are restricted to local variables only. M1 Build supports use of enumerated data types for channels and parameters, as they provide more information than a Boolean data type.

String

String data types are restricted to local variables only. They can be used to show text in information windows that can open up in M1 Tune.

Properties of an Object

Properties include:

- Old Name
- Quantities and Base Units
- Display Defaults
- Validation
- Storage Classes.

Old Name

An alternative name for the object can be defined. If, during migration in M1 Tune, an object with the same name cannot be found, M1 Tune looks for an object that has the Old Name in the Package to migrate from.

This allows for an automatic migration from old Packages in M1 Tune even when objects have been renamed.

Quantities and Base Units

Every floating point channel, parameter and table must be assigned a quantity. The quantity defines the physical property that the object represents, for example temperature or mass.

Each quantity defines a base unit which is the unit in which the value of the object is stored.

If the **base unit** is not suitable for data entry, a **display unit** can be chosen. Conversion from display unit to base unit is automatically handled by the M1 System.

EXAMPLE

The **base unit** for pressure is the Pascal (Pa). If a calculation is performed in a function which adds two pressures with **displayed** values of 100psi and 80bar, the performed mathematical calculation is:

689475.729Pa (100psi) + 8000000Pa (80bar) = 8689475.729Pa

The result of which could then be displayed as kilopascals (kPa), resulting in a displayed value of 8689.475729kPa.

If a channel, parameter or table is configured with a data type of integer, unsigned integer or enumeration the only available quantity is **unitless**.

All calculations in the M1 System are performed in base units. The base units used within the M1 System are generally SI units, exceptions are listed below.

Current exceptions to SI units in the M1 System		
Quantity	**M1 Base Unit**	**SI Unit**
Angle	Degrees	Radians
Temperature	Celsius	Kelvin

Display Defaults

Display defaults define the default settings used to display an object.

All calculations are performed using the base units for the selected quantity, but display of an object will use display defaults.

☛ *Display settings have no influence on the internal value of an object.*

Display settings can be overridden by M1 Tune.

Display Unit

The Display Unit is the unit in which an object is displayed. This setting can be overridden by M1 Tune.

> EXAMPLE
>
> A voltage channel can be displayed in millivolts or kilovolts.

Display Format

The Display Format changes the format in which the object is displayed. This setting can be overridden by M1 Tune.

> EXAMPLE
>
> A Display Format of 'hexadecimal' can be used for CAN IDs.
>
> A Display Format of 'hours, minutes' can be used for engine run time.

☛ *Alternate display formats for time are not used for data entry.*

Decimal Places (DPS)

Defines how many decimal places are shown by default when an object is displayed. This setting can be overridden by M1 Tune.

Minimum and Maximum

Defines the default scaling used when an object is displayed in M1 Tune. For example, a channel representing engine temperature could have minimum set to 50 degrees and maximum to 150 degrees.

Validation

Validation limits define the allowable range of values for an object. Limits on calibration objects are enforced by M1 Tune.

Limits can also be defined for data objects. These limits are not enforced by M1 Build as assignment to data objects is performed by the M1 ECU firmware.

If an application requires limits to be enforced on a data object there are options such as:

- Use the "Limit.Range" function, see the Library Functions topic.
- Use the "Validate()" method or the "Constrain()" method on the data object, see the Methods topic.

EXAMPLE

An example of validation usage is in a duty cycle channel.

A duty cycle cannot vary outside of a 0% to 100% range. So in this instance, it is sensible to set a validation range on the channel of 0% to 100%.

Storage Classes

An object's storage class defines the way it is stored within the M1 ECU. Storage classes apply to data objects only.

There are two storage classes:

- Volatile
- Flash backed

Volatile

This is the default storage class.

An object stored in volatile storage does not retain its value when the M1 ECU firmware stops. Each time the M1 ECU firmware starts, the object is initialised to zero.

An initial value can be set for a volatile object by using a scheduled function associated with the startup event. The initial value can be a constant, a literal or a parameter.

EXAMPLE

A channel representing a sensor value is usually stored in volatile storage.

This is because a new sensor value is available immediately after the M1 ECU firmware starts.

Flash Backed

An object can be stored in flash backed storage at the request of the M1 ECU firmware. Each time the M1 ECU firmware starts, the object's value is initialised from the previously stored data.

EXAMPLE

A channel representing "Fuel Used" is usually stored in flash backed storage. This is because after restart of the M1 ECU firmware, a new value of "Fuel Used" cannot be computed from current sensor inputs. A historic value is required to continue the calculation.

☛ *If there is no previously stored value available, the object is initialised to zero.*

All flash backed objects are not stored permanently, but only when calling the "System.Preserve" library function. The maximum storage rate is 1Hz.

Scheduling

Most objects in a Project require methods to be periodically executed in order to function. Each of these methods must have an event associated with it so that M1 Build can appropriately schedule it during the compilation process.

Events can only be used if they have been added to the Project.

M1 Build automatically determines the correct order of execution by performing a dependency analysis on the Project, trying to establish a reasonable process flow leading from acquiring input values to determining output values. The function call order is displayed in the 'Schedule' section of M1 Build (see the M1 Build User Manual for details). If M1 Build is unable to determine the execution order a conflict is flagged with an icon. The most common error to occur during scheduling is a circular dependency. In general circular dependencies can be resolved by restructuring the offending code. A circular dependency often indicates an error on the part of the application designer.

Procedures to avoid circular dependencies include separating inputs from outputs by handling them in different functions. Also using the GetUnscheduled() method on channels can detach a channel from the dependency analysis in M1 Build.

If the Project requires a given order of function execution, it should be verified that the scheduling is still the same after changes have been made in the Project, especially when adding or removing channels that are written and read in different functions.

The M1 Programming Language

The M1 programming language is a language similar to C customised for simple integration into the M1 Build environment.

General Syntax

General syntax usage covered in this section include:

- A comparison to C.
- Assignment of variables

Comparison to C

Many constructs are similar to C.

Braces { } are used to define blocks of code.

Parentheses ( ) are used to define precedence and clarify expressions.

Like in C, all statements are terminated by a semicolon ";".

After a closing brace } no semicolon is required.

All conditions are subject to short circuit evaluation.

Both C and C++ style comments are supported:

```
/* this is a c style comment */
// this is a c++ style comment
```

However, some C features are not supported, such as:

- referencing, dereferencing and indirection
- pointers
- arrays
- some integer and floating point data type variants, see Data Types
- iteration statements like 'do', 'while', 'for' loops
- some operators like logical negation "!", equality "==="
- dynamic memory allocations

Additionally, boolean and string data types are only allowed for local variables.

Assignment of variables

A channel can only be assigned once per code path. This means that it is not possible to assign the same channel in two different functions or twice during the execution of a function.

EXAMPLES

A channel named Target is to be assigned in a function. Code of the following form is not allowed:

```
Target = Parameter 1;
if (condition)
{
    Target = 600.0;
}
```

As Target can be assigned twice when executing the function.

The correct construct to use here is 'if...else':

```
if (condition)
{
    Target = 600.0;
}
else
{
    Target = Parameter 1;
}
```

A local variable does not have any restrictions on assignment.

An alternative solution to the previous example is:

```
local interim = Target; /* initialise local variable */
interim = Parameter 1;
if (condition)
{
   interim = 600.0;
}
Target = interim;
```

Constructs

Constructs covered in this section are:

- if...else
- when...is
- expand...to
- local/static local

if...else

The `if` statement tests the parenthesised condition, and if the condition is true, executes the group of statements in the braces that follow.

```
if ([condition])
{
   ...
}
```

EXAMPLES

Some `if...else` usage examples:

```
if (Channel A > 100 and Channel A < 150)
{
   /* Channel A is greater than 100 and less than 150 */
   ...
}
else if (Channel A neq Channel B or Channel A eq Channel C)
}
   /* Channel A is not equal to Channel B or Channel A is equal to Channel C */
   ...
}
else if (Channel D >= 0 and Channel D <= 99)
{
   /* Channel D is greater than or equal to 0 and less than or equal to 99 */
   ...
}
else if (Channel E eq true and Channel F eq false)
{
   /* Channel E is true and Channel F is false. */
   ...
}
else
{
   ...
}
```

when...is

The `when` keyword begins a `when/is` construct. The `when/is` construct takes the following form:

```
when ([argument])
{
   is ([enumerator])
   {
      ...
   }
   is ([enumerator] or [enumerator])
   {
      ...
   }
}
```

The [argument] used in the `when` statement must be of an enumerated data type.

Each [enumerator] must be one of the enumerators of the enumeration, and all of the enumerators of the enumeration must be covered by the `when/is` construct.

The `or` keyword can optionally be used to specify multiple enumerators to match an `is` statement.

EXAMPLES

How to use a `when/is` construct:

Channel A is of an enumerated data type with enumerators OK, Warning and Error.

```
when (Channel A)
{
    is (OK)
    {
        /* Do something */
    }
    is (Warning or Error)
    {
        /* Do something */
    }
}
```

expand...to

This keyword marks the beginning of an expand statement. The expand statement takes the following form:

```
expand ([name] = [start] to [end])
{
    /* This block expanded multiple times. */
    ...
}
```

An `expand` statement performs inline expansion of the following block. `[start]` and `[end]` must be literals or constants, and represent integer values that are 0 or positive.

Each time the block is expanded `$(name)` is replaced by the current value of `[name]`, and `[name]` is incremented by 1.

This substitution is a text substitution only. The expand statement does not verify the generated code.

`$(name)` can be expanded anywhere in the code. Example usages are object names and literals.

EXAMPLES

Some `expand...to` usage examples:

Assign the value of a table named 'Table A' to four channels named 'Channel 1' to 'Channel 4':

```
expand (N = 1 to 4)
{
    Channel $(N) = Table A;
}
```

This will be expanded to the following:

```
Channel 1 = Table A;
Channel 2 = Table A;
Channel 3 = Table A;
Channel 4 = Table A;
```

Assign the value of a table named 'Table A' to a number of channels. The number of channels is stored in a constant object named 'Engine Cylinders':

```
expand (N = 1 to Engine Cylinders)
{
    Channel $(N) = Table A;
}
```

local/static local

The `local` keyword is used to define a local variable. Local variables are used in functions and are not available outside of the containing function. Also they cannot be seen or logged in M1 Tune. Inside the function, a local variable can only be used after it has been defined.

A local variable must be assigned an initial value.

```
local [name] = [value];
```

By default the data type of the local variable will be the same as the data type of the value initially assigned to it.

The data type of the local variable can optionally be explicitly stated by:

```
local <[Data Type]> [name] = [value];
```

A local variable will be created and assigned to its initial value each time a function is executed and discarded when the function returns. To preserve the value of a local variable between executions of a function use the `static` keyword.

When using the `static` keyword, the first time a function is executed the local variable is assigned its initial value. For all subsequent executions of the function the local will contain the value stored during the previous execution of the function.

```
static local [name] = [value];
static local <[Data Type]> [name] = [value];
```

☛ *The initial value of a local variable can come from any value provider, and the local variable will be of the same data type as the initialiser. In contrast to this, the initial value of a **static** local variable **must** be a literal value, enumerator or constant value.*

EXAMPLES

Some variable examples:

```
local example = 1.0;
```

will create a local variable example with the data type floating point and initialise the local variable with a value of 1.0.

```
local example = 1;
```

will create a local variable example with the data type signed integer and initialise the local variable with a value of 1.

```
local <Unsigned Integer> example = 1;
```

will create a local variable example with the data type unsigned integer and initialise the local variable with a value of 1.

```
local example = Switch State.On;
```

will create a local variable example with the enumerated data type 'Switch State' and initialise the local variable with the enumerator 'On'.

```
local example = Channel A;
```

will create a local variable example with the same data type as Channel A and initialise the local variable with the current value of Channel A.

```
local example = Delay.Rising(Channel A > 1.0, 5.0);
```

will create a local variable example with the data type boolean, and will assign the result of the calculation of the library function to this value. Note that in this case the internal values of the library function are maintained between executions of the function.

Operators

Operators include:

- Floating Point Arithmetic
- Integer Arithmetic
- Enumeration Comparison
- Floating Point Comparison
- Integer Comparison
- Logical
- Integer Bitwise
- Compound assignment
- Other

All supported operators are equivalent to those in the C programming language. Details on the function of these operators can be found in a standard C programming manual.

Floating Point Arithmetic

Name	Syntax
Assignment	`a = b`
Addition	`a + b`
Subtraction	`a - b`
Arithmetic Negation	`-a`
Multiplication	`a * b`
Division	`a / b`

Integer Arithmetic

Name	Syntax
Assignment	`a = b`
Addition	`a + b`
Subtraction	`a - b`
Arithmetic Negation	`-a`
Multiplication	`a * b`
Division	`a / b`
Modulo	`a % b`

Enumeration Comparison

Name	Syntax
Equal to	`a eq b`
Not equal to	`a neq b`

☛ *`eq` is equivalent to the C operator `==`, and `neq` is equivalent to the C operator `!=`.*

Floating Point Comparison

Name	Syntax
Greater than	`a > b`
Less than	`a < b`
Greater than or equal to	`a >= b`
Less than or equal to	`a <= b`

☛ *Equality and inequality tests on floating point values are not supported.*

Integer Comparison

Name	Syntax
Equal to	`a eq b`
Not equal to	`a neq b`
Greater than	`a > b`
Less than	`a < b`
Greater than or equal to	`a >= b`
Less than or equal to	`a <= b`

☛ *`eq` is equivalent to the C operator `==`, and `neq` is equivalent to the C operator `!=`.*

Logical

Name	Syntax
Logical negation (NOT)	`not a`
Logical AND	`a and b`
Logical OR	`a or b`

☛ *These operators are equivalent to the C alternative tokens for `!`, `&&` and `||` respectively.*

Integer Bitwise

Name	Syntax
Bitwise NOT	`~a`
Bitwise AND	`a & b`
Bitwise OR	`a | b`

Name	Syntax
Bitwise XOR	`a ^ b`
Bitwise left shift	`a << b`
Bitwise right shift	`a >> b`

Compound Assignment

Name	Syntax
Addition assignment	`a += b`
Subtraction assignment	`a -= b`
Multiplication assignment	`a *= b`
Division assignment	`a /= b`
Modulo assignment	`a %= b`
Bitwise AND assignment	`a &= b`
Bitwise OR assignment	`a |= b`
Bitwise XOR assignment	`a ^= b`
Bitwise left shift assignment	`a <<= b`
Bitwise right shift assignment	`a >>= b`

Other

Name	Syntax
Comma (as a separator of arguments)	`a, b`
Function call	`function(argument)`
Ternary conditional	`a ? b : c`
Member b of object a	`a.b`

Keywords

This section explains keywords used in M1 Build.

In

The `In` keyword refers to an object which stores the input arguments to a function. These arguments can then be referenced using the '.' operator.

EXAMPLE

In this example an input argument named 'Filter Constant' is referenced:

```
local fc = In.Filter Constant;
```

Out

The `Out` keyword refers to an object which stores the return value of a function. The return value can be assigned using the = operator.

EXAMPLE

In this example the function returns a value that is stored in the local named 'result' :

```
local result = 100;
Out = result;
```

Parent

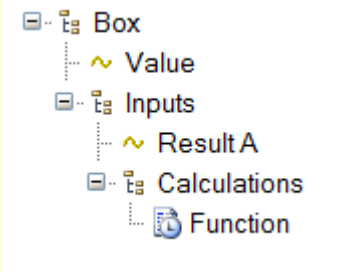
The `Parent` keyword forms part of an object reference. The parent of an object 'x' is the object containing 'x'.

When used without any preceding qualifications the `Parent` keyword will resolve to the parent of the group that the current object is stored in.

The `Parent` keyword can be chained using the '.' operator.

EXAMPLES

In this example a function is stored in a group named 'Calculations', this group is stored within another group named 'Inputs'. The group 'Inputs' contains a channel called 'Result A'.



The following example shows a reference to 'Result A' using the `Parent` keyword. The value of Result A is stored in a local variable 'result' in the function. Only a single `Parent` keyword is required in this case as the reference refers to the group that the function is stored within.

No explicit references to the groups 'Calculations' or 'Inputs' are required:

```
local result = Parent.Result A;
```

The following example shows a reference to 'Value' which is stored another level higher:

```
local result = Parent.Parent.Value;
```

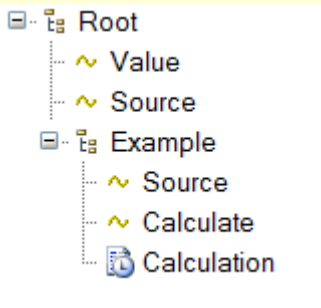
Root

The `Root` keyword forms part of an object reference. Root is the root group of a Project and is therefore the first constituent of an absolute object reference.

The `Root` keyword can be used to resolve ambiguous references where an object of the same name exists within different groups.

EXAMPLES

In this example, a Project contains the following objects: a channel named 'Source' in the root group and a channel of the same name in a group named 'Example'. The group also contains a scheduled function 'Calculation'.



This example demonstrates an unambiguous reference to the channel 'Source' from the root group in the function:

```
local result = Root.Source;
```

Library

The `Library` keyword forms part of a library function reference.

The `Library` keyword can be used to resolve ambiguous references when there is a conflict between an object name and a library function name.

EXAMPLE

In this example a project contains a channel named 'Calculate' and a scheduled function in the same group.

In the function, the following code references a library function called 'Calculate.Max()':

```
local result = Library.Calculate.Max(Value, 1.0);
```

This

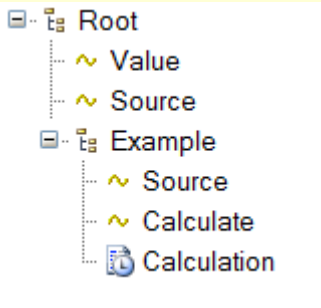
The `This` keyword forms part of an object reference.

`This` refers to the Group that the current object is stored within.

The `This` keyword can be used to resolve ambiguous references where an object of the same name exists within different groups or an object name conflicts with a library function name.

EXAMPLES

In this example a project contains the following objects: a channel named 'Source' in the root group and a channel of the same name in a group named 'Example'. The group also contains a channel named 'Calculate' and a scheduled function 'Calculation'.



The following example demonstrates an unambiguous reference to the channel 'Source' from the group 'Example' in the function:

```
local result = This.Source;
```

The following example demonstrates an unambiguous reference to the channel 'Calculate', as there exist library functions using the same name.

```
local result = This.Calculate;
```

False

Represents the logical value `false`.

Used in conditional expressions and when passing logic values to functions and methods.

For an example, see the if...else topic.

True

Represents the logical value `true`.

Used in conditional expressions and when passing logic values to functions and methods.

For an example, see if...else topic.

Assignments

An assignment statement sets the value of a channel or a local variable.

In general, the expression of the assignment must have the same data type as the variable.

EXAMPLES

A channel 'Engine State' has the data type 'Engine State Enumeration' with the enumerators:

```
-1 = Error
0 = Stopped (default)
1 = Cranking
2 = Idle
3 = Running
4 = Overrun
```

Therefore, Engine State can only be assigned with expressions that are of the same data type:

```
Engine.State = Engine State Enumeration.Idle;
```

Enumerators can also be assigned by using the associated Set()-method of the enumeration:

```
Engine.State.Set(Engine State Enumeration.Idle);
```

or equivalent by using the enumerator value if the enumerator Idle:

```
Engine.State.Set(2);
```

However, numeric values, which are represented using a floating point, integer or unsigned integer data type, are handled differently.

M1 Build performs conversions of integer data types in assignments and calculations if they involve different numeric data types. The expression is spread from left to right, evaluating each step of the expression separately by converting operands to a floating point value if one of the operands is a floating point value, and if not, converting operands to an unsigned integer value if one of the operands is an unsigned integer value.

If the expression contains floating point values, it must be assigned to a floating point variable.

If the expression does not contain floating point values, the assigned variable can be of any of the numeric data types.

EXAMPLES

Using two local variables:

```
local <Unsigned Integer> low = 5;
local high = 6.5;
```

a valid assignment is:

```
high = low;
```

as 'low' can be converted into a floating point data type.

A valid assignment is also:

```
high = low * (-7);
```

this is because '-7' (which is a signed integer) can be converted into an unsigned integer (to match 'low'), and the result can be converted into a floating point data type.

However, the internal conversion of (-7) into an unsigned integer will result in a very high value of 'high' due to the bitwise interpretation of integer values (refer to respective publications for more details on bit operations). Therefore, expressions with different integer data types should be avoided, or the library functions **Convert.ToInteger()** or **Convert.ToUnsignedInteger()** should be used.

The following expression is not valid:

```
low = high;
```

as a conversion of 'high' to an integer data type is not allowed. The **Convert.ToUnsignedInteger()** function must be used in this case.

Comparisons

The result of a comparison is a boolean value, which is either true or false.

Only enumerated data types or integer values can be compared on equality or inequality with the `eq` or `neq` operators.

EXAMPLE

Using the previous example for the channel 'Engine State' in the Assignments topic, a possible comparison is:

```
if (Engine State eq Engine State Enumeration.Idle)
{
...
}
```

Floating point values are scaled dependent on their range. Therefore, especially as a result of calculations, floating point values may show deviations from the expected value in the least significant decimal places as a result of rounding to the nearest representable value.

EXAMPLE

A value of 0.0027 is assigned to a floating point channel 'Value'. Next, 0.00005 is added six times to this variable, and then the variable is compared with a threshold to generate an action:

```
if (Value >= 0.003)
{
    // do something spectacular
    ...
}
```

However, when executing this code, nothing happens. The reason for this is that Value is in fact not exactly 0.003, but 0.0029999993275851. Also, the defined compare value of 0.003 shows a rounding error in its floating point implementation and has a value of 0.00300000002607703.

Therefore, floating point comparisons should be regarded carefully, keeping the rounding effects in mind (refer to respective publications for more details on floating point operations).

One possibility to evaluate if a floating point value is almost equal to another value is:

```
Value = 0.03;
if (Calculate.Absolute(Value - 0.03) < 0.0001)
{
    // do something
    ...
}
```

However, the compare value above (0.0001) must be chosen equally careful based on the anticipated value ranges.

In general, only objects of the same data type can be compared.

However, numeric values, which are represented using a floating point, integer or unsigned integer data type, are handled differently. Both integer data types can be compared with a floating point value. In this case, when executing the comparison, the integer value will be converted into an internal floating point value first, then the comparison is performed.

☛ *Comparisons between unsigned and signed integer values are not allowed.*

The options shown below are available to convert different data types for comparisons.

An enumerated object's integer value can be accessed using the `AsInteger()` method.

EXAMPLE

Using the channel 'Engine State' (used in previous examples) to determine if an engine is not stopped or has an error, a comparison could check if the enumerator value is greater than '0':

```
if (Engine State.AsInteger() > 1)
{
...
}
```

The library function `Convert.ToInteger()` allows to convert a floating point value or an unsigned integer value to the closest integer value.

The library function `Convert.ToUnsignedInteger()` allows to convert a floating point value or an integer value to the closest unsigned integer value.

Some conversions of numeric data types can also be realised by using assignments, see Assignments topic.

Function Calls

User defined functions and library functions follow the same syntax.

Function calls take a form where optional arguments are passed to the function and an optional result is returned. In addition, library functions may have several overloads which can support different data types, or add additional arguments.

Arguments may be passed as individual objects or complex expressions. Expressions are evaluated at the time the function is executed. The result of the expression is passed as the respective argument. Equivalent conversions as for assignment operations are also performed.

Syntax:

```
Function(Argument1, Argument2)
```

OR

```
Function(Argument1, Argument2, Argument3)
```

If the function returns a result, the function call must be part of an expression.

EXAMPLE

In the following example the function returns a value which is assigned to the channel LeanerBank.

```
LeanerBank = Calculate.Max(Lambda.Left.Bank, Lambda.Right.Bank);
```

In the next example, the user defined function Clear() is called and Position is passed as an argument. The function does not return a result.

```
Clear(Position);
```

In the last example, the function is evaluated and the result is compared to Lambda.Aim, returning true or false within the if statement.

```
if (Calculate.Average(Lambda.Left.Bank, Lambda.Right.Bank) > Lambda.Aim)
{
}
```

Library Functions

Library functions are provided by M1 Build to perform common calculations and interact with the underlying operating system. For example, a library function could:

- Transmit and receive CAN messages
- Calculate the value of mathematical functions (sin, square root, etc.)
- Decode messages from an external device (GPS, etc.).

☛ *Some library functions (filters, etc.) maintain internal state, for example if they perform comparisons between a current and a previous condition.*

These functions cannot be called conditionally or used within comparisons, but must be called on every execution of a scheduled function or method and are specially annotated in M1 Build by a purple function icon Change.By

Validation errors are reported if these functions are used within a conditional statement.

Some functions deal with more than one data type (for example Floating Point or Integer), and results are returned accordingly. These cases are shown in the following tables with OR in the Syntax column to distinguish variants of the function call.

In all of the following examples the data which is passed to the function is termed Argument, Argument1, Argument2 etc. The return value is not shown as it is implicit in the function call process.

Calculate

The calculate library functions are provided to perform common arithmetic operations. They include statistical, trigonometric and time based functions. There is also a collection of simple helper functions to aid readability. Each function may have several overloads which can support different data types, or add additional arguments.

Calculate Function	Description	Arguments	Returns	Syntax
Calculate.Absolute	Returns absolute value of argument	Integer or Floating Point	Integer or Floating Point	Calculate.Absolute(Argument)
Calculate.Average	Returns average value of two arguments	Integer or Floating Point	Integer or Floating Point	Calculate.Average(Argument1, Argument2)
Calculate.Between	Returns Boolean True if argument is between range [min, max] for at leastfilter time.	Arguments - Integer or Floating Point, Filter time - Floating Point	True or False	Calculate.Between(Argument, min, max, filter)
Calculate.Beyond	Returns Boolean True if argument is beyond range [min, max] for at least filter time.	Arguments - Integer or Floating Point, Filter time - Floating Point	True or False	Calculate.Beyond(Argument, min, max, filter)

Calculate Function	Description	Arguments	Returns	Syntax
Calculate.Bias	Returns biased average of two arguments. Bias ranges from -1 (minimum of a and b) to 0 (average of a and b) to +1 (maximum of a and b)	Arguments - Integer or Floating Point, Bias - Floating Point	Floating Point	Calculate.Bias(Argument1, Argument2, bias)
Calculate.Ceiling	Rounds argument to the next higher integer value	Floating Point	Floating Point	Calculate.Ceiling(Argument)
Calculate.FastCos	Returns Cosine of argument (<0.5% error)	Floating Point	Floating Point	Calculate.FastCos(Argument)
Calculate.FastSin	Returns Sine of argument (<0.5% error)	Floating Point	Floating Point	Calculate.FastSin(Argument)
Calculate.FastSquareRoot	Returns square root of argument (<0.0013% error)	Floating Point	Floating Point	Calculate.FastSquareRoot(Argument)
Calculate.FastTan	Returns Tangent of argument (<0.5% error)	Floating Point	Floating Point	Calculate.FastTan(Argument)
Calculate.Floor	Rounds argument to the next lower integer value	Floating Point	Floating Point	Calculate.Floor(Argument)
Calculate.Hysteresis	Returns boolean true if argument is >= high limit for filter time, false if argument is <= low limit for filter time.	Arguments - Integer or Floating Point, Filter time - Floating Point	True or False	Calculate.Hysteresis(Argument, Low, High, Filter)
Calculate.Infinity	Returns a Value that represents positive infinity	-	Infinity	Calculate.Infinity()
Calculate.InverseCos	Returns Inverse Cosine of argument	Floating Point	Floating Point	Calculate.InverseCos(Argument)
Calculate.InverseSin	Returns Inverse Sine of argument	Floating Point	Floating Point	Calculate.InverseSin(Argument)
Calculate.InverseTan	Returns Inverse Tangent of argument	Floating Point	Floating Point	Calculate.InverseTan(Argument)
Calculate.InverseTan2	Returns Inverse Tangent including quadrant of argument	Floating Point	Floating Point	Calculate.InverseTan2(ArgumentY, ArgumentX)
Calculate.IsNAN	Tests whether the argument is not a number	Floating Point	True or False	Calculate.IsNAN(Argument)
Calculate.Max	Returns the greater of two arguments	Integer or Floating Point	Integer or Floating Point	Calculate.Max(Argument1, Argument2)
Calculate.MaximumFloat	Returns the maximum floating point value	-	The maximum floating point value	Calculate.MaximumFloat()
Calculate.Min	Returns the lesser of two arguments	Integer or Floating Point	Integer or Floating Point	Calculate.Min(Argument1, Argument2)
Calculate.Modulo	Returns the remainder of Argument / Denominator	Integer or Floating Point	Integer or Floating Point	Calculate.Modulo(Argument, denom)

Calculate Function	Description	Arguments	Returns	Syntax
Calculate.NAN	Returns a value that is "not a number"	-	"not a number"	Calculate.NAN()
Calculate.PI	Returns constant value of PI	-	Floating Point	Calculate.PI()
Calculate.Power	Calculate x to the power of y	Floating Point	Floating Point	Calculate.Power(Argument x, Argument y)
Calculate.Stable	Returns Boolean True if argument has not changed for Filter seconds	Arguments - Integer or Floating Point, Filter time - Floating Point	True or False	Calculate.Stable(Argument, Filter)

CanComms

CanComms functions provide CAN messaging control and status information.

CanComms Function	Description	Arguments	Returns	Syntax
CanComms.BusStatus	Returns enumerated Status of specified bus.	Integer	Enumerated Status Value	CanComms.BusStatus(bus)
CanComms.GetBit	Read one Bit from message at bit offset 0 to 63.	Handle, Integer	Bit	CanComms.GetBit(handle, bitoff)
CanComms.GetFixed7DP	Read fixed point 7DP number from CAN message.	Handle, Bitoff	Fixed point	CanComms.GetFixed7DP(handle, bitoff)
CanComms.GetFloat	Read 32-bit IEEE754-2008 floating point number from CAN message.	Handle, Bitoff	Floating point	CanComms.GetFloat(handle, bitoff)
CanComms.GetID	Extract message ID from current CAN receive message	Handle	Integer	CanComms.GetID(handle)
CanComms.GetInteger	Extract a signed integer of size bitlen (1 to 32) from offset bitoff (0 to 63) in current CAN receive message.	Integers	Integer	CanComms.GetInteger(handle, bitoff, bitlen)
CanComms.GetLength	Returns message length in bytes of specified handle	Handle	Integer	CanComms.GetLength(handle)
CanComms.GetTicks	Get message receive time in ticks	Handle	Unsigned Integer	CanComms.GetTicks(handle)
CanComms.GetUnsignedInteger	Extract an unsigned integer of size bitlen (1 to 32) from offset bitoff (0 to 63) in current CAN receive message.	Integers	Integer	CanComms.GetUnsignedInteger(handle, bitoff, bitlen)
CanComms.Init	Initialise CAN Bus bus at kbaud kilobits/second for communications	Integers	True or False	CanComms.Init (bus, kbaud)
CanComms.RxMessage	Receive CAN message from handle.	Handle	True or False	CanComms.RxMessage (handle)

CanComms Function	Description	Arguments	Returns	Syntax
CanComms.RxOpenExtended	Open a CAN receive handle on Bus (0-2), at 23-bit Address match, with address Mask mask, word alignment true for Bigendian.	Integers, Boolean	Handle	CanComms.RxOpenExtended(bus, match, mask, bigendian)
CanComms.RxOpenStandard	Open a CAN receive handle on Bus (0-2), at 11-bit address Address match, with address mask Mask, word alignment true for Bigendian.	Integers, Boolean	Handle	CanComms.RxOpenStandard(bus, match, mask, bigendian)
CanComms.SetBit	Set one Bit in message at offset 0 to 63	Handle, Integer, Boolean		CanComms.SetBit(handle, bitoff, bitval)
CanComms.SetFloat	Write 32-bit IEEE754-2008 floating point number into CAN message.	Handle, Bitoff, Val		CanComms.SetFloat(handle, bitoff, value)
CanComms.SetInteger	Populate a signed integer of size bitlen (1 to 32) from offset bitoff (0 to 63) in current CAN transmit message.	Integers		CanComms.SetInteger(handle, bitoff, bitlen, value)
CanComms.SetUnsignedInteger	Populate an Unsigned integer of size bitlen (1 to 32) from offset bitoff (0 to 63) in current CAN transmit message.	Integers		CanComms.SetUnsignedInteger(handle, bitoff, bitlen, value)
CanComms.TxExtended	Transmit message from current handle on bus Bus at 23-bit address ID.	Handle, Integers	True or False	CanComms.TxExtended(handle, bus, ID)
CanComms.TxExtendedRR	Transmit CAN remote request frame with extended id	Integers	True or False	CanComms.TxExtendedRR(bus, ID, len)
CanComms.TxInitialise	Clear message pointed to by current Handle prior to populating with desired transit values. Length lenspecifies number of bytes to clear.	Handle, Integer		CanComms.TxInitialise(handle, len)
CanComms.TxOpen	Open a Transmit CAN handle, Bigendian byte order if True	Boolean	Handle	CanComms.TxOpen (bigendian)
CanComms.TxStandard	Transmit message from current handle on bus Bus at 11-bit address ID, returns true is message was successfully transmitted.	Handle, Integers	True or False	CanComms.TxStandard(handle, bus, ID)
CanComms.TxStandardRR	Transmit CAN remote request frame with standard id	Integers	True or False	CanComms.TxStandardRR(bus, ID, len)
CanComms.XOR8	Calculate exclusive or of all bytes in message	Handle		CanComms.XOR8(handle)

Using the CanComms Functions

For Startup

Typically, on Startup, the following is required to enable CAN operation:

- A CAN Bus class must be present.
- This class runs CanComms.Init() on Startup and then CanComms.BusStatus at 5Hz.

☛ *While these library functions may be called manually this is not recommended, manual calling may be removed in future versions.*

To Receive CAN Messages

Typically, to receive CAN messaging, the following is required:

A function must be executed to receive all messages at the composite CAN receive rate. Four addresses at 50Hz would require an execution speed of at least 200Hz. The function does the following:

- A local variable is declared to open a CAN receive handle. This handle specifies the bus, the address, the address mask, and the word order (bigendian or not). This instruction is only executed on the first call to one of the Open functions.
- The handle is examined to see if a matching CAN message has been received.
- If so, data is extracted from the message identified by the handle.
- If more than one CAN address is to be received, multiple local handles are declared.

☛ *CAN Timeouts are not integrated into library functions. They must be manually programmed.*

EXAMPLE

As follows:

```
local h = CanComms.RxOpenStandard(0, 0x123, 0x0, true);
if (CanComms.RxMessage(h))
{
    Cooldown.Timer = CanComms.GetUnsignedInteger(h, 32, 16);
}
```

To Transmit CAN Messages

Typically, to Transmit CAN messaging, the following is required:

- Execute a function at the desired CAN transmit rate. This does the following:
  - A local variable is declared to open a CAN transmit handle.
  - The transmit message buffer for the new handle is cleared.
  - The transmit message buffer for the new handle is populated with transmit data.
  - The transmit message buffer is transmitted on a specified bus and address.
- Update status condition based on the boolean result from the Transmit function.

EXAMPLE

As follows:

```
local h = CanComms.TxOpen(true);
CanComms.TxInitialise(h, 8);
CanComms.SetUnsignedInteger(h, 0, 16, Convert.ToUnsignedInteger(Engine.Speed / 36.0));
CanComms.SetUnsignedInteger(h, 16, 16, Convert.ToUnsignedInteger(Throttle.Pedal * 1.0e3));
if (CanComms.TxStandard(h,0,0x100))
{
    TX Status = TX Status.Ok;
}
else
{
    TX Status = TX Status.Fault;
}
```

Change

These functions return boolean **true** or **false** values determined by whether their argument has changed in a given fashion.

Change Function	Description	Arguments	Returns	Syntax
Change.By	Returns Boolean True if argument has changed by Delta for Filter Time.	Arguments - Integer or Floating Point, Filter time - Floating Point	true or false	Change.By(Argument, delta, filter) **OR** Change.By(Argument, delta)
Change.Down	Returns Boolean True if argument has changed downwards by Delta for Filter Time.	Arguments - Integer or Floating Point, Filter time - Floating Point	true or false	Change.Down(Argument, delta, filter) **OR** Change.Down(Argument, delta)
Change.Either	True for one iteration after at least **filter** seconds from **cond** changing	Condition - Boolean, Filter Time - Floating Point	true or false	Change.Either(cond, filter) **OR** Change.Either(cond)
Change.From	True for one iteration after at least **filter** seconds from **cond** becoming false..	Condition - Boolean, Filter Time - Floating Point	true or false	Change.From(cond, filter) **OR** Change.From(cond)
Change.To	True for one iteration after at least **filter** seconds from **cond** becoming true.	Condition - Boolean, Filter Time - Floating Point	true or false	Change.To(cond, filter) **OR** Change.To(cond)
Change.Up	True for one iteration after at least **filter** seconds from **arg** becoming greater than its previous value by **delta**	Arguments - Integer or Floating Point, Filter time - Floating Point	true or false	Change.Up(Argument, delta, filter) **OR** Change.Up(Argument, delta)

Convert

The convert functions convert a value from one data type to another.

Convert Function	Description	Arguments	Returns	Syntax
Convert.ToInteger	Round to nearest Integer value.	Floating Point or Unsigned Integer	Integer	Convert.ToInteger(Argument)
Convert.ToUnsignedInteger	Round to nearest Unsigned Integer value.	Floating Point or Integer	Unsigned Integer	Convert.ToUnsignedInteger(Argument)

Debounce

Debounce functions return **true** or **false** values determined by whether their argument has remained stable for the filter time, which commences upon a change of state.

Debounce Function	Description	Arguments	Returns	Syntax
Debounce.Fast	Debounce **cond** for at least **filter** seconds.	Boolean and Floating Point	true or false	Debounce.Fast(cond, filter)
Debounce.Filter	Debounce boolean Condition with a Smoothing filter and hysteresis using 0.2 and 0.8 thresholds.	Boolean and Floating Point	true or false	Debounce.Filter(cond, response)
Debounce.Stable	Debounce **cond** for at least **filter** seconds. If **cond** reverts to original state filtering is restarted.	Boolean and Floating Point	true or false	Debounce.Stable(cond, filter)
Debounce.Verify	Debounce **cond** for at least **filter** seconds. Verify that **cond** is in new state after filtering.	Boolean and Floating Point	true or false	Debounce.Verify(cond, filter)

Delay

Delay functions return **true** or **false** values after a delay defined by the delay time.

The Syntax is similar to the Calculate functions.

Delay Function	Description	Arguments	Returns	Syntax
Delay.Falling	The argument **cond** must be false for at least **delay** seconds before the return value transitions to false.	Boolean	Boolean	Delay.Falling(cond, delay)
Delay.Rising	The argument **cond** must be true for at least **delay** seconds before the return value transitions to true.	Boolean	Boolean	Delay.Rising(cond, delay)
Delay.Signal15	Delay signal by up to 15 samples	Floating Point	Delayed	Delay.Signal15(argument, delay)
Delay.Stable	The argument **arg** must be stable for at least **delay** seconds before the return value transitions to true.	Argument: Floating Point or Integer Delay, Delta: Floating Point	Boolean	Delay.Stable(argument, delay, delta) **OR** Delay.Stable(argument, delay)

Derivative

Derivative functions offer three forms.

Derivative Function	Description	Arguments	Returns	Syntax
Derivative.Normal	Calculate the normal derivative.	Floating Point	Floating Point	Derivative.Normal(Argument)
Derivative.Filtered	Calculate the filtered derivative.	Floating Point	Floating Point	Derivative.Filtered (Argument)

Derivative Function	Description	Arguments	Returns	Syntax
Derivative.Adaptive	Calculate the adaptive filtered derivative of Argument, using minimum Delta and maximum time max_dt before derivative update.	Floating Point	Floating Point	Derivative.Adaptive (Argument, delta, max_dt)

Filter

Filter functions replicate commonly used filter models. The various forms are detailed below.

Filter Function	Description	Arguments	Returns	Syntax
Filter.FirstOrder	Value of Argument is first order filtered with time constant tc, optional reset function.	Floating Point and Boolean	Floating Point	Filter.FirstOrder(Argument, tc) **OR** Filter.FirstOrder(Argument, tc, reset)
Filter.Maximum	Tracks a higher current value of Argument and filters towards a smaller current value of Argument with first order filter and optional reset.	Floating Point and Boolean	Floating Point	Filter.Maximum(Argument, tc) **OR** Filter.Maximum(Argument, tc, reset)
Filter.Minimum	Tracks a smaller current value of Argument and filters towards a greater current value of Argument with first order filter and optional reset.	Floating Point and Boolean	Floating Point	Filter.Minimum(Argument, tc) **OR** Filter.Minimum(Argument, tc, reset)

First Order

First order digital filter of the form

```
y[n] = a0 * x[n] + b1 * y[n - 1],
```

Where the filter coefficients a0 and b1 are calculated such that the filter has the time constant tc. To elaborate: $a0 = \Delta T/(tc+\Delta T)$; $b1 = 1 - a0$; where ΔT is 1/fs with fs being the sample rate in Hz.

If the reset input is true, the filter is reset with the value of x[n] (the current passed value to filter).

The range of time constants depends on the sample rate of the filter. Recommended time constants fall in the range (1/fs, 1/fs * 100) seconds. There is no upper limit on tc, but calculation accuracy may be reduced for excessively large values.

The filter's characteristics should not change if its sample rate is changed unless the limits described above are encountered.

Maximum

Tracks the maximum and filters towards the current passed value to filter with a first order digital filter as described under First Order.

The filter will be reset each time the reset input is true or the passed value to filter is greater than the current filter value.

Minimum

Tracks the minimum and filters towards the current passed value to filter with a first order digital filter as described under First Order.

The filter will be reset each time the reset input is true or the passed value to filter is smaller than the current filter value.

Integral

Integral Function	Description	Arguments	Returns	Syntax
Integral.Normal	Returns the integral of argument, clamped within Min and Max, with optional Reset function and Preset value.	Floating Point and Boolean	Floating Point	Integral.Normal (Argument, min, max, reset, preset)

Limit

Limit functions return the argument clamped to max, min, or both values.

Limit Function	Description	Arguments	Returns	Syntax
Limit.Max	Returns Value limited to Max.	Integer or Floating Point	Integer or Floating Point	Limit.Max(Argument, max)
Limit.Min	Returns Value limited to Min.	Integer or Floating Point	Integer or Floating Point	Limit.Max(Argument, min)
Limit.Range	Returns Value limited within Min and Max.	Integer or Floating Point	Integer or Floating Point	Limit.Range(Argument, min, max)

Logging

Logging functions return the status of the Logging system.

Logging Function	Description	Arguments	Returns	Syntax
Logging.Running	Returns true if Logging is running, or if specified logging system (0 - 7) is running.	Integer	true or false	Logging.Running() **OR** Logging.Running(system)
Logging.Unloading	Returns true if Logging is unloading, or if specified logging system (0 - 7) is unloading.	Integer	true or false	Logging.Unloading() **OR** Logging.Unloading(system)
Logging.Used	Returns Ratio of flash memory used in specified system (0 - 7)		Floating Point	Logging.Used(system)

Serial

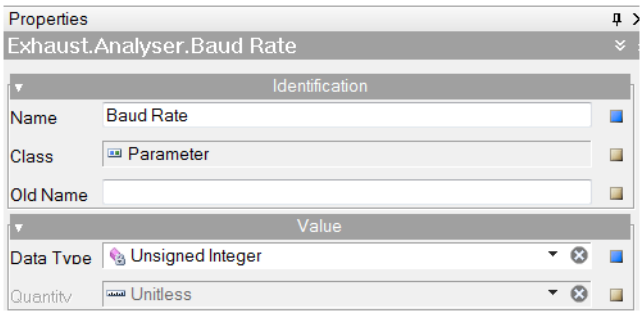
Serial Function	Description	Arguments	Returns	Syntax
Serial.GetHandle	Open a serial handle	True or False	Handle	Serial.GetHandle(bigendian)
Serial.GetInteger	Read signed integer	Handle, Offset, Integer	Integer	Serial.GetInteger(handle, offset, length)
Serial.GetLinOffset	Find the data offset for a LIN frame with the specified ID	Handle, LIN ID		GetLinOffset(handle, id)
Serial.GetTransmitHandle	Open transmit mode serial handle DEPRECATED: use GetHandle	True or False	Handle	Serial.GetTransmitHandle(bigendian)
Serial.GetUnsignedInteger	Read unsigned integer	Handle, Offset, Integer	Integer	Serial.GetUnsignedInteger(handle, offset, length)
Serial.LinDump	Print all packets to the ECU log	Handle		Serial.LinDump(handle)
Serial.PortDiagnostic	Check serial port diagnostic	Port	Port	Serial.PortDiagnostic(port)
Serial.PortInit	Initialise serial port for communications	Port,	True or False	Serial.PortInit(port, baud, protocol)
Serial.Receive	Receive data from the serial port	Handle, Port	True or False	Serial.Receive(handle, port)
Serial.SetFloat	Write floating point value into serial message	Handle, Offset, Value	Offset	Serial.SetFloat(handle, offset, value)
Serial.SetFloat	Write integer into serial message	Handle, Offset, Integer	Offset	Serial.SetInteger(handle, offset, length, value)
Serial.SetInteger	Write integer into serial message	Handle, Offset, Integer	Offset	Serial.SetInteger(handle, offset, length, value)
Serial.SetLinHeader	Write a LIN header into the serial message	Handle, Offset, LIN, ID, True or False	Offset	Serial.SetLin.Header(handle, offset, length, id, tx, enhanced)
Serial.SetString	Write a string into serial message	Handle, Offset, Length, String	Offset	Serial.SetString(handle, offset, length, text)
Serial.SetUnsignedInteger	Write unsigned integer into serial message	Handle, Offset, Integer	Offset	Serial.SetUnsignedInteger(handle, offset, length, value)
Serial.Sum8	Calculate sum of the message data bytes	Handle, Offset, Integer		Serial.Sum8(handle, offset, length)

Serial Function	Description	Arguments	Returns	Syntax
Serial.Transmit	Send len bytes on serial port	Handle, Port, Length		Serial.Transmit(handle, port, length)
Serial.XOR8	Calculate exclusive-or of the message data bytes	Handle, Offset, Length		Serial.XOR8(handle, offset, length)

Using the Serial Functions

For Startup

- For Serial functions to operate an RS232 class must be present.
- A parameter must be present which defines the Baud Rate.



- A Startup function must initialise the RS232 subsystem. The following example may be used:

EXAMPLE
```
if (Serial.PortInit(0,Baud Rate,0))
{
Diagnostic = Diagnostic.OK;
}
else
{
Diagnostic = Diagnostic.Configuration Error;
}
```

To Receive RS232 Messages

As RS232 is a data transfer standard which does not define exact message formatting, individual requirements must be handled in scheduled functions. At present it is not possible to define framing of RS232 messages, other than to clear the receive buffer and then examine the received message byte by byte in order to determine the framing byte position(s).

Typically, to receive RS232 messages, the following is required:

A function must be executed to read the port and make use of the received data. A basic example follows. This function would be scheduled at roughly the repetition rate of the receive message.

EXAMPLE

```
// read serial port
local h = Serial.GetHandle(true);
local Byte 0 = 0x0;
local Byte 1 = 0x0;
local Byte 2 = 0x0;
local Byte 3 = 0x0;
local Byte 4 = 0x0;
local Byte 5 = 0x0;
local Byte 6 = 0x0;
local Byte 7 = 0x0;

if (Serial.Receive(h,0))
{
Byte 0 = Serial.GetUnsignedInteger(h,0,1);
Byte 1 = Serial.GetUnsignedInteger(h,1,1);
Byte 2 = Serial.GetUnsignedInteger(h,2,1);
Byte 3 = Serial.GetUnsignedInteger(h,3,1);
Byte 4 = Serial.GetUnsignedInteger(h,4,1);
Byte 5 = Serial.GetUnsignedInteger(h,5,1);
Byte 6 = Serial.GetUnsignedInteger(h,6,1);
Byte 7 = Serial.GetUnsignedInteger(h,7,1);
}
```

Each time the function is called the buffer is read and flushed.

To Transmit RS232 Messages

A variable length transmit buffer must first be populated, and then transmitted. In the following example the hexadecimal values 0x1B, 0x30, 0x35, 0x0D are transmitted.

EXAMPLE

```
local txbuffer = Serial.SetUnsignedInteger(h,0,1,0x1B);
txbuffer = Serial.SetUnsignedInteger(h,1,2,0x3035);
txbuffer = Serial.SetUnsignedInteger(h,3,1,0x0D);
Serial.Transmit(h,0,4);

txbuffer = txbuffer;
```

System

System Function	Description	Arguments	Returns	Syntax
System.AllowTuning	Enable or Disable Calibration modifications.	Boolean		System.AllowTuning (value)
System.CpuIdle	The proportion of time the cpu is idle		Floating Point (Ratio)	System.CpuIdle()
System.Debug	Write a message to the ECU system log with an optional value.	String and Floating Point		System.Debug(msg) **OR** System.Debug(msg, value)

System Function	Description	Arguments	Returns	Syntax
System.ElapsedTime	Returns elapsed time in seconds since current line of code was executed.		Floating Point	System.ElapsedTime()
System.EventMiss	Returns the number of times this thread has missed an event.		Unsigned Integer	System.EventMiss()
System.FlashSize	Returns the size of Flash Memory		Unsigned Integer	System.FlashSize()
System.HiResTickPeriod	Returns the high resolution tick period in seconds.		Floating Point	System.HiResTickPeriod()
System.HiResTicks	Returns the high resolution tick count (<10 seconds)		Unsigned Integer	System.HiResTicks()
System.HiResTicksSince	Returns the elapsed high resolution ticks since the supplied tick count.		Unsigned Integer	System.HiResTicksSince(prev)
System.Preserve	Forces a preserve of all Flash Memory backed channels.			System.Preserve()
System.RamSize	Returns the size of RAM Memory		Unsigned Integer	System.RamSize()
System.SerialNumber	Returns the ECU Serial Number		Unsigned Integer	System.SerialNumer()
System.TickPeriod	Returns the system tick period in seconds.		Floating Point	System.TickPeriod()
System.Ticks	Returns the current System Tick count		Unsigned Integer	System.Ticks()
System.TicksBetween	Returns the number of ticks between the Begin and End ticks	Unsigned Integer	Unsigned Integer	System.TicksBetween (begin, end)
System.TicksRemaining	Returns the number of ticks from Ticks to Limit	Unsigned Integer	Unsigned Integer	System.TicksRemaining(ticks, limit)
System.TicksSince	Returns the number of ticks since Previous	Unsigned Integer	Unsigned Integer	System.TicksSince(prev)
System.TimedDebug	Write a message to the ECU system - limited to one message per second	String		System.TimedDebug(msg)
System.XcpConnected	Returns true if Tune software is connected to ECU.		true or false	System.XcpConnected()

Other (Integrated) Functions

Following are library functions that may be integrated into classes, and if so, should not be used directly:

- BR2
- GPS
- LTC
- MDD
- PDM
- RS232Comms
- VCS

Library Functions for Calibration Methods

All the library functions mentioned in this reference cannot be used within calibration methods. Calibration methods provide their own set of library functions that can only be used within calibration methods. These functions include:

- Maths functions
- System functions
- UI functions

Calibration Maths Functions

Math Function	Description	Arguments	Returns	Syntax
Math.Abs	Returns absolute value of argument	Integer or Floating Point	Integer or Floating Point	Math.Abs(Argument)
Math.Acos	Returns the arccosine in radians	Floating Point	Floating Point	Math.Acos(Argument)
Math.Asin	Returns the arcsine in radians	Floating Point	Floating Point	Math.Asin(Argument)
Math.Atan	Returns the arctangent in radians	Floating Point	Floating Point	Math.Atan(Argument)
Math.Cos	Returns the cosine in radians	Floating Point	Floating Point	Math.Cos(Argument)
Math.Cosh	Returns the hyperbolic cosine in radians	Floating Point	Floating Point	Math.Cosh(Argument)
Math.Exp	Returns the exponential (natural e)	Floating Point	Floating Point	Math.Exp(Argument)
Math.Exp10	Returns the exponential (base 10)	Floating Point	Floating Point	Math.Exp10(Argument)
Math.IsNaN	Tests whether the argument is not a number	Floating Point	True or False	Math.IsNAN(Argument)
Math.Log	Returns the logarithm (natural base)	Floating Point	Floating Point	Math.Log(Argument)
Math.Log10	Returns the logarithm (base 10)	Floating Point	Floating Point	Math.Log10(Argument)
Math.Max	Returns the greater of two arguments	Integer or Floating Point	Integer or Floating Point	Math.Max(Argument1, Argument2)
Math.Min	Returns the lesser of two arguments	Integer or Floating Point	Integer or Floating Point	Math.Max(Argument1, Argument2)
Math.Sgn	Returns the sign of the argument	Integer or Floating Point	-1, 0 or+1	Math.Sgn(Argument)
Math.Sin	Returns the sine in radians	Floating Point	Floating Point	Math.Sin(Argument)
Math.Sinh	Returns the hyperbolic sine in radians	Floating Point	Floating Point	Math.Sinh(Argument)
Math.Sqr	Returns square of argument	Integer or Floating Point	Integer or Floating Point	Math.Sqr(Argument)
Math.Sqrt	Returns square root of argument	Floating Point	Floating Point	Math.Sqrt(Argument)
Math.Tan	Returns the tangent in radians	Floating Point	Floating Point	Math.Tan(Argument)
Math.Tanh	Returns the hyperbolic tangent in radians	Floating Point	Floating Point	Math.Tanh(Argument)

Calibration System Functions

System Function	Description	Arguments	Returns	Syntax
System.Debug	Writes a message to the ECU system log	String	-	System.Debug(msg)
System.StrCat	Joins two strings together	String	String	System.StrCat(Argument1, Argument2)

Calibration UI Functions

UI Function	Description	Arguments	Returns	Syntax
UI.PromptOK	Displays a dialog box	String	-	UI.PromptOK(Argument1 title, Argument2 message)

Programming

This topic serves the purpose of giving suggestions on how to create a structured, consistent and straightforward Project with M1 Build.

Adhering to a certain programming standard has many advantages. It:

- Encourages a focus on content rather than layout
- Allows one to understand the Project more quickly by making assumptions based on previous experience
- Makes it easier to modify and add to a Project
- Reduces the possibilities of errors.

It is therefore highly recommended that in every Project a certain, defined standard is maintained.

☛ ***Though it is not compulsory to apply the following suggestions, they represent recommendations that have proven advantageous when using M1 Build.***

Project Structure

There are some boundary conditions to consider when fixing the Project structure.

The objects in a Project are scoped by their containing group. That means an object can only be accessed when the group of its origin is known and addressed. Inside a group, all objects need to have different names. But they can have the same names as another object in another group.

EXAMPLE

Within the 'Root' group, there is a group named 'Engine' and a group named 'Vehicle'. Both may contain a channel named 'Speed'. If any function wants to use the engine speed information, it is necessary to have the reference of the desired channel, and in this case the channel to use would be (when using an absolute reference):

```
(Root.)Engine.Speed
```

Even when all objects in all groups are given a unique name, M1 Build does not provide the possibility to access an object only by its name.

On the other side, this boundary condition of M1 Build allows to design the Project easily in a logical tree structure.

On the main level ('Root'-group), main groups should be defined according to the logical main tasks and elements as well as output features the Project has to comprise.

EXAMPLE

In an engine Project, main groups would include groups such as:

```
Engine
Fuel
Ignition
Torque
...
```

Inside each group, a further split into logical objects can be realised.

EXAMPLE

In the main group 'Fuel' subgroups could comprise groups such as:

```
Pressure
Fuel
Timing
Injector
...
```

Also the related hardware should be included in the main groups. For example in the main group 'Fuel' also the fuel injector control should be included. So the subgroup 'Injector' may for example include a channel 'Pulse Width' which is used as output onto the injector.

☛ *The name of this channel, including the group reference, would be Fuel.Injector.Pulse Width, which is self documenting.*

☛ *See Assignment of variables topic. As explained in that topic, a channel can only be assigned once in an iteration which has to be realised unambiguously. This also forces the realisation of a clear logical Project structure, where each channel has a distinct place where it belongs to and can be assigned.*

The advantage of the structure explained above allows:

- The work flow inside a Project is easily comprehensible
- Separation of concerns can be realised
- Object names already indicate their use.

To use a Project structure in the described way is therefore highly recommended.

However, it can become difficult to maintain the integrity of a Project in cases such as:

- Large-scale Projects
- Numerous programmers working on one Project
- Where high modularity with defined interfaces is desired.

Given the difficulties imposed in such cases, it can be helpful to add interface groups to each main group where all input and output items are uniquely defined, and other groups are only allowed to use these interface items.

This allows development groups independence from the rest of the Project. However, it does require a clear administration of the interface items and organisation and maintenance of the overall Project composition.

Naming Conventions

Names should be unambiguous and not the same as the name of a class or function name or part thereof, as otherwise problems may occur in maintaining references when objects are moved or edited. Also, this means not to differentiate names only by using uppercase/lowercase characters.

Naming Objects

- Begin name with an uppercase letter
- Space may be used between two name constituents
- Keep names short and simple
- Name according to the meaning of the object

Additionally, if a task-based, modular structure is used in the Project, one should take advantage of the fact that objects are referenced by their parent directory.

EXAMPLE

A channel which conveys the fuel pressure needs to be named only 'Pressure', when included in a main group named 'Fuel', as its referenced name is then

```
Fuel.Pressure
```

Naming Local Variables

- Begin name with a lowercase letter (camelCase is optional)
- No spaces should be used
- Don't distinguish local variable names from other object names only by uppercase/lowercase writing (problems with maintaining references will occur)

Easily differentiating locals variables and channels assists in readability.

EXAMPLE

As follows:

```
local fuelvol = 3.1;
local pulseWidth = fuelvol * Fuel.Injector.Flow;
Fuel.Injector.Pulse Width = pulseWidth;
```

Naming Enumerators

- Names should be kept short to improve readability in M1 Tune. The Help text of an enumerator can be used to provide more information.
- The default value for all data types in a Project is Zero. So assigning a value of Zero to a certain enumerator means that all objects using this enumeration will be initialised with the enumerator that corresponds to the enumerator value of Zero when the M1 is activated. Therefore the enumerator corresponding to a neutral or initial condition should be assigned with a value of Zero.
- Enumerators can have negative or positive enumerator values. Enumerators that correspond to errors should be assigned to negative values, and enumerators that occur during normal operation should be assigned to positive values, facilitating their use in code.

Code Layout and Format

Consider the following:

- Write only one statement per line
- Write only one declaration per line
- Align braces with keyword
- Use a separate line for each brace
- Use parentheses to clarify clauses in an expression
- Use parentheses for conditions
- Put a space between a keyword and a parenthesis
- Don't put a space between a function and a parenthesis

EXAMPLE

```
if ((a > b) and (b < c))
{
    somevalue = 2;
}
Library.Calculate.Max(a, b);
```

- Declare local variables in the most constrained scope
- All code begins in the first column
- No trailing whitespace
- Indentation by tab character only
- All functions and methods to end with a blank line
- Avoid magic numbers. Rather, refer to a constant which describes the purpose of the number
- Do not put spaces in front of semicolons.

- Indent subsequent lines by one tab stop
- Indent conditional block by one tab stop
- If a line gets too long, split statement into two lines.

EXAMPLE

As follows:

```
if (condition1 and condition2 and condition3
    and condition4)
{
    somevalue = 42;
}
else if (condition2)
{
    somevalue = 1;
}
else
{
    somevalue = 2;
    if (condition3)
    {
        othervalue = 7;
    }
}
```

Ternary conditional ? :

- Put the condition in parentheses
- Put the action for the 'then' and 'else' statement on a separate line if one line makes it unclear.

EXAMPLE

As follows:

```
local x = (condition) ? 1 : 2;
local y = (condition)
    ? Library.Calculate.Max(a, b)
    : Library.System.Ticks();
```

Tags

Each object in an M1 Build Project can be assigned a set of tags. Tags assigned to a group node are inherited by the children of the group. Tags are used for filtering and searching in both M1 Build and M1 Tune. The available tags are predefined and fall into three groups:

1. **System**
The system group describes the functional system of the object. There are three possible tags:
- Engine: Objects related to the engine
- Vehicle: Objects related to the vehicle
- Driver: Objects related to the driver

If a tag in this group is not selected, M1 Build will emit a warning.

2. **Type**
The type group describes the type of object. There are six possible tags:
- Normal: Channels which are of primary interest and should be normally visible. This includes important measurements such as "Coolant Temperature" and "Engine Speed".
- Diagnostic: Channels which contain diagnostic information.
- Advanced: Channels which are rarely of interest. This includes internal parts of calculations and rarely used measurements.
- Pin: Channels which contain values directly related to physical pins on the M1 ECU. This includes voltage measurements and output duty cycles.
- Tune: Tables and parameters which are usually adjusted while the engine is running. This includes "Engine Efficiency" and "Ignition Timing Main".
- Setup: Tables and parameters which are usually adjusted when configuring the Package. This includes "Engine Cylinders" and "Wheel Speed Front Circumference".

If a tag in this group is not selected M1 Build will emit a warning.

3. **I/O**
The I/O group describes the I/O direction of the object. This tag is optional. There are two possible tags:
- Input: Objects which are related to inputs to the M1 ECU. This includes "Coolant Temperature" and "Inlet Manifold Pressure".
- Output: Objects which are related to outputs of the M1 ECU. This includes "Fuel Pump" and "Warning Light".

Code Commenting

Good code is partially self documenting. Avoid redundant comments.

Use sufficient comments, especially where code is complex and/or not self-explaining.

Use comments for structure if a function contains several sections.

However, keep the code tidy and clear.

EXAMPLES

As follows:

```
/*
* A 'C' style block comment
*/
/* A 'C' style single line comment */
// A C++ style comment
```

Project Help

It is highly recommended to make good use of the Help-section in the Properties-window. All help that is filed there will be accessible for the user in M1 Tune and can facilitate usage of the Package. Therefore, the Help should comprise:

- Channels: purpose and information given by the channel and where it is used. Note that M1 Tune automatically adds the enumeration to the Help if the channel is of an enumerated data type.
- Parameter and table: purpose and calibration information such as recommended values or consequences of certain values. Note that M1 Tune automatically adds the enumeration to the Help if the parameter is of an enumerated data type.
- Groups: content of the group and basic information about connections, calculations and dependencies inside of the group, including interactions with other main groups. If a group has a default value, the group help should refer to the default value.

By using the provided text styles in the M1 Build Help editor, the help content can be formatted and structured to support a consistent appearance in M1 Tune. In addition, Information and Warning styles provide the ability to further emphasize important help content. See the M1 Build User manual for details.

☛ *It is good practice to write the Help immediately when creating an object. Expecting to write all Help at the one time after finishing a project often is not realised, as time pressures toward the end of a project prohibit spending a significant amount of time on help.*

Correcting Code and Handling Compiler Errors

It is recommended to regularly use the M1 Build Validate Package option to check the correctness of the Project, as it may get difficult to sort out all errors when many changes were done between two checks.

The Validate Package option will save the Project and check if the current Project is able to generate valid code and show the result of this check in the Messages window.

If an error is displayed, an explanation and the way to correct it are shown by using the Show error details option in the Messages menu. By double-clicking on the error or the displayed location, the object or code line causing the error is selected in the main window.

Some errors may be consequential errors, reported as a result of a single error. Therefore, it is recommended to search the error list for the following errors first:

- Object or local '*xxxx*' does not exist
- Syntax Error : Expecting ';'
- Syntax Error : Unexpected symbol '*xxxx*' in statement - are you missing a ';' or '}' ?
- Syntax Error : Unexpected symbol ';' in conditional - are you missing a ')' ?
- Any other kind of 'Syntax Error'
- Wrong Data Type errors.

It is recommended first to correct these errors and then validate the Package again. For the correct syntax, refer to the displayed error details and see the respective topic in this manual.

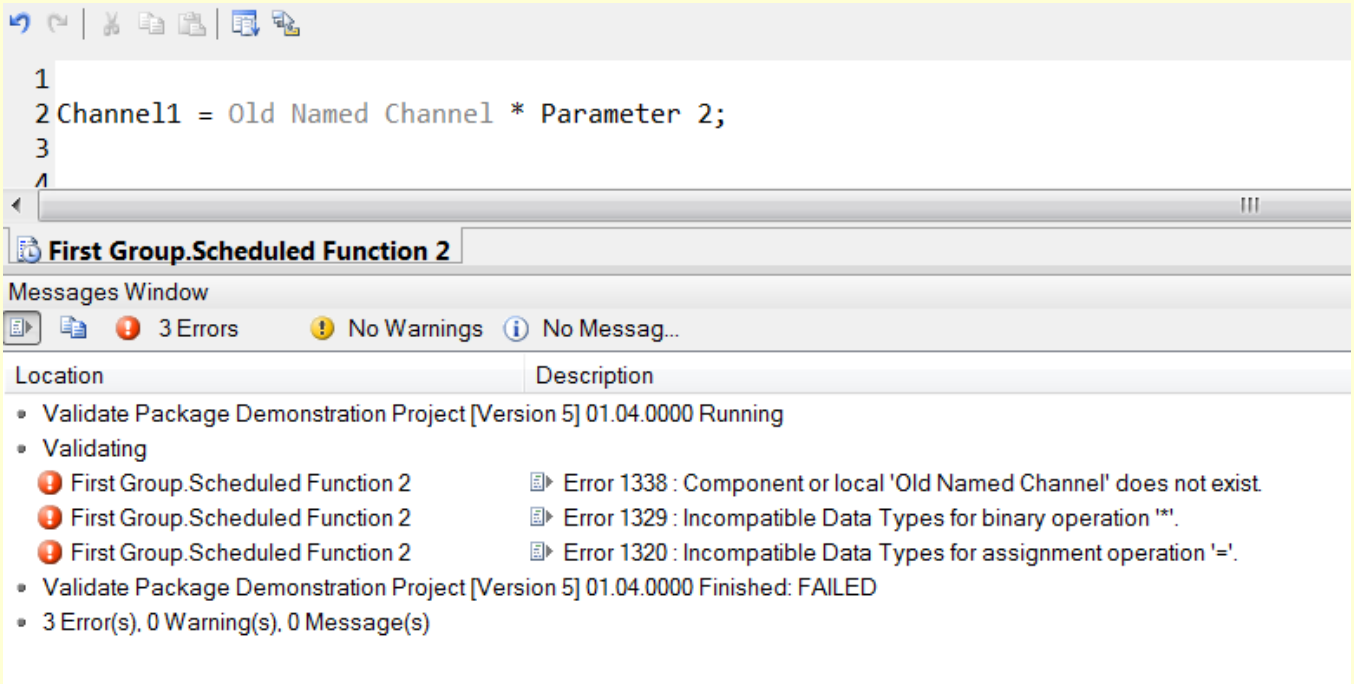
If still errors remain, check for errors that indicate non existing items or wrong data types, correct these errors and then validate the Package again.

Usually after doing so, no consequential errors remain and all errors that still exist have to be corrected by themselves. Please read the advice on how to correct the error by choosing the 'Show error details' option, and additionally the relevant chapter in this manual.

☛ *In rare cases, checking the code by validating the Package will not show an error, but when building a Package an error occurs during the compilation process. Show error details will then not deliver the location of the error inside of the Project, but inside the generated code. This should be reported to MoTeC (using Send Feedback... in the Help-menu of M1 Build).*

EXAMPLE

In the following example, the channel 'Old Named Channel' does not exist in the Project, but the function tries to use such a channel. The respective error message is the first one in the list, and the following error messages are consequential errors only.



EXAMPLE

In the following example, a semicolon has been forgotten after a statement in the function 'Injectors.calc'. This error is usually shown in the lower part of the error list, as shown below.

Messages Window

14 Errors | No Warnings | No Messag...

Location	Description
Validate Package TestBenchReplacementM800Dash [Inputs on M1 Switches revised] 01.23.0005 Running	
Validating	
Engine.Speed.Reference.Test Speed.Value	Error 1627 : Channel value not assigned.
Injectors.State	Error 1627 : Channel value not assigned.
Injectors.Diagnostic	Error 1627 : Channel value not assigned.
Injectors.Engine Speed Manual	Error 1631 : Parameter value not read.
Injectors.Engine Speed Mode	Error 1631 : Parameter value not read.
Injectors.Engine Speed	Error 1627 : Channel value not assigned.
Injectors.Speed Change Indicator	Error 1627 : Channel value not assigned.
Injectors.Output Trigger Value	Error 1627 : Channel value not assigned.
Injectors.Passing	Error 1627 : Channel value not assigned.
Injectors.Valid Offtime	Error 1627 : Channel value not assigned.
Injectors.Offtime	Error 1631 : Parameter value not read.
Injectors.Maximal Duty Cycle	Error 1631 : Parameter value not read.
Injectors.calc	Error 10000 : Syntax Error : Expecting ';'
Injectors.calc	Error 1025 : Script parse error.
Validate Package TestBenchReplacementM800Dash [Inputs on M1 Switches revised] 01.23.0005 Finished: FAILED	
14 Error(s), 0 Warning(s), 0 Message(s)	

All other errors in the above list are consequential errors and vanish after the semicolon had been added in line 39 of the function. The line number is shown as indicated below when the error details are shown.

Injectors.Maximal Duty Cycle	Error 1631 : Parameter value not read.
Injectors.calc	**Error 10000 : Syntax Error : Expecting ';'**
Injectors.calc	Location : Scheduled Function 'Injectors.calc' (39)
Validate Package TestBenchReplacementM800Das[illegible]ILED	
14 Error(s), 0 Warning(s), 0 Message(s)	

The error list usually looks similar when other syntax errors have been made, such as if braces have been forgotten or keywords have been misspelled.

Security

The MoTeC M1 ecosystem includes a comprehensive security system based on public-private key cryptography. The security system protects the Package created by M1 Build. The M1 Build Project and source code are not protected.

The security system supports multiple user accounts; however, only one user can be logged in at a time.

Security is strictly enforced. MoTeC cannot recover lost credentials, and cannot recover data if it is protected. If access to an ECU is lost MoTeC can clear the contents of the ECU to restore access; however, all user data will be destroyed.

The Package files stored by M1 Tune are protected by the same security algorithms used by the M1 ECU. If a Package has security enabled, opening the Package in M1 Tune requires the same credentials as logging in to the ECU. This means that if credentials are lost, the Package files cannot be opened or recovered.

There are three ways to configure security:

- **Off**
  Security is completely disabled and all objects are freely accessible.
- **Basic**
  The security system is automatically configured such that users can be added using M1 Tune.
  This configuration allows an end-user of the Package to define one or more users with protected access to all data and calibration objects.
- **Advanced**
  The author of the M1 Build Project has complete control of security settings. Each data and calibration object in the Project must be assigned to a security group. Users and log systems can then be defined with access to these security groups.
- **Automatic**
  The security groups in the M1 Build Project are automatically configured according to their selected tags. Each data and calibration object in the Project is assigned to a security group consisting of the combination of the tags selected on the object. Users and log systems can then be defined with access to these security groups.

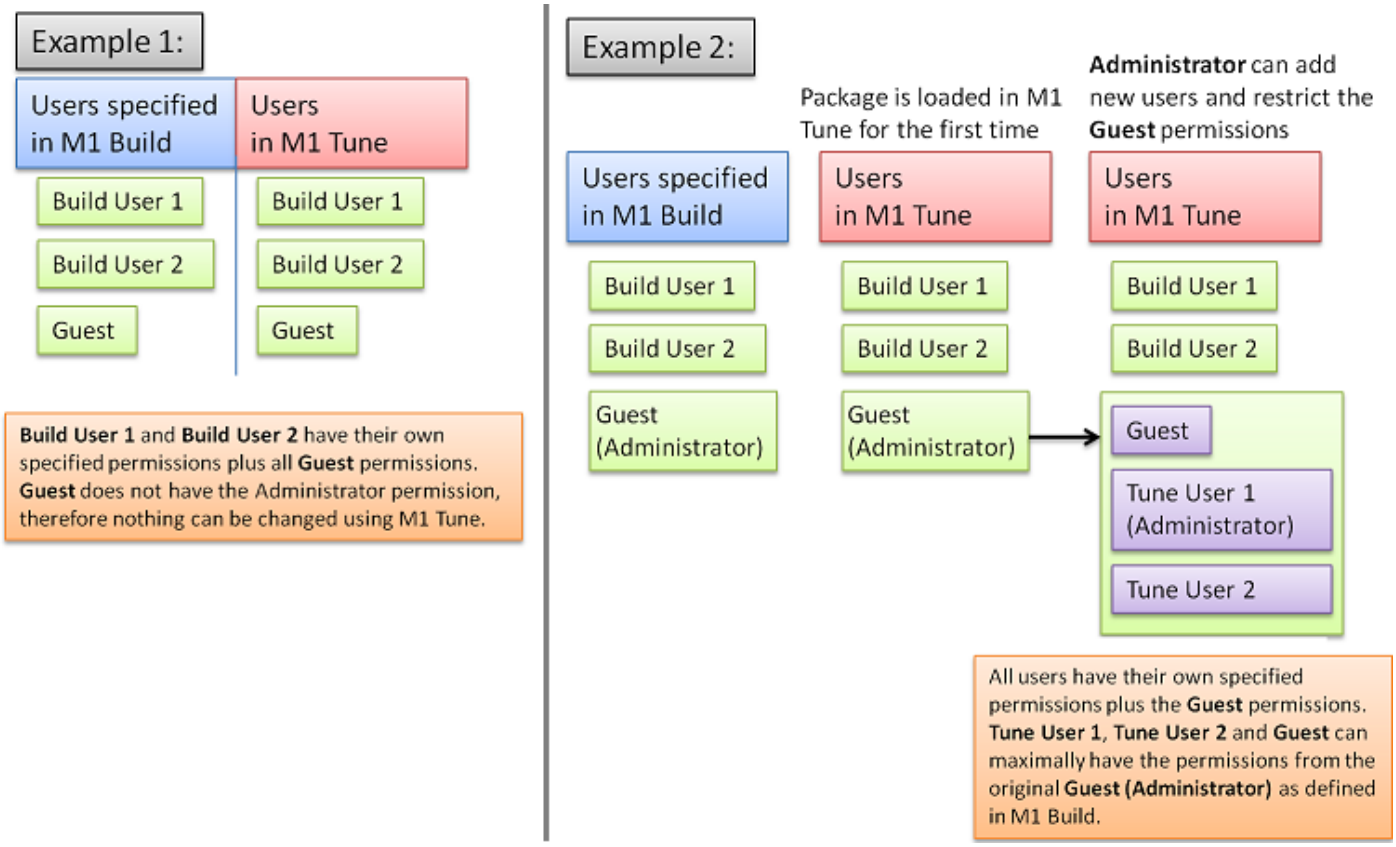
Security Model

The security system has two parts, the configuration defined in M1 Build and the configuration defined in M1 Tune.

The M1 Build configuration is hard coded into the Package and cannot be modified using M1 Tune. The 'Guest' user contained in the M1 Build configuration defines the maximum level of access available to M1 Tune users.

To further restrict access to the Package, the 'Guest' user can be overridden when M1 Tune users are created.

Security Model



Users

Each user has a name, credentials and permissions. A user's credentials can be either a password or a key file. The user's permissions define the access rights that the user has to the data contained within the ECU and the Package.

Users can be defined in M1 Build and in M1 Tune.

Users created using M1 Build

When a user is defined in the M1 Build Project it is immutable. This means that the user's credentials and permissions cannot be subsequently changed using M1 Tune.

Users defined in this way are most commonly for the purpose of protecting critical data or intellectual property from end-users of the Package.

Users created using M1 Tune

Any user with the 'Administrator' permission can create and destroy users using M1 Tune. An administrator can grant permissions to other M1 Tune users up to their own level of access.

The administrator cannot modify users created using M1 Build.

The 'Guest' user

The 'Guest' user is a special user created automatically by M1 Build. It requires no credentials to login.

The guest user determines two important aspects of the security configuration:

- Whether users can be created using M1 Tune.
- The maximum access available to users created using M1 Tune.

If the guest user has been granted the 'Administrator' permission, users can be created using M1 Tune. Once a new administrator has been created, the guest user can be modified to reduce its level of access. See above examples.

The permissions granted to the guest user apply to all users.

Security Groups

A security group is a collection of data and calibration objects. Security groups are defined using M1 Build and cannot be modified after the Package has been built.

Each data and calibration object in the Project must be assigned to a security group and can only be a member of one group at a time.

At least one user must have 'Adjust' access to each security group.

Up to 64 security groups can be configured.

Log Systems

Log systems have a similar security configuration to a user. This allows a log system to be restricted to only log a particular set of data objects.

If a user can convert a log system, they must also have access to read the channels logged by that system.

Permissions

Administrator

The 'Administrator' permission controls access to the credentials and permissions of other users. This applies to users created using M1 Tune only and cannot be granted to a user created using M1 Build, apart from the 'Guest' user.

Allow Send Firmware

The 'Allow Send Firmware' permission can take three values:

- Yes
  Any firmware can be sent by the user.
- No
  No firmware can be sent by the user.
- Upgrade Only
  The user can only send a newer version of the firmware currently running in the ECU.

Allow Change Password

The user is allowed to change their password. This applies to users created using M1 Tune only.

Allow Change Workbooks

The user is allowed to change the workbooks stored in the ECU.

Allow Change ECU Name

The user is allowed to change the name of the ECU.

The ECU name is advertised over the network and shown by M1 Tune when prompting for a connection.

Tuning

For each memory group, the user has three possible access levels:

- Adjust
  The user can view data and adjust calibration contained within the security group.
- View
  The user can view data and calibration contained within the security group. The user cannot make adjustments to calibration contained within the security group.
- None
  The user has no access to the security group.

Data Logging

There are a number of user permissions for each data logging system. These permissions control how the user can interact with the data logging system.

- Allow Access
  Similar to memory group permissions, there are three possible access levels:
  - Adjust
    The user can make changes to the logging configuration.
  - View
    The user can view but not adjust the logging configuration.
  - None
    The user has no access to the logging configuration.
- Allow Retrieval
  The user can retrieve logged data from the ECU. This does not mean that the user can view the contents of the logged data.
- Allow Convert
  The user can convert retrieved logged data to a format suitable for use with MoTeC i2. This does not mean that the user can retrieve the logged data.
- Allow Erase
  The user can erase logged data from the ECU.

Logging

The M1 allows to record data channels internally with up to 8 logging systems. The general capabilities of the logging systems are dependent on the available logging memory (dependent on the M1 hardware type) and the ECU license which defines the maximum allowed recording rate of the logging system. With logging level 1, the maximum rate is 10Hz. Level 2 allows up to 200Hz and with level 3 a rate of 1000Hz is supported.

The start and stop conditions of each logging system, the logged channels and their log rate are configurable, however logging system 1 automatically contains the channels specified for 'Diagnostic Logging' in M1 Build. Also access to certain logging systems and/or channels can be restricted using 'Advanced' security settings.

In addition to the available logging memory the recording duration is determined by the number of chosen channels and their log rate. Cyclic logging allows to always keep the latest data. When connected to M1 Tune the logged data can be downloaded and converted for further analysis, for example with i2.

Different aspects of the logging system setup are handled in M1 Tune and M1 Build.

In M1 Tune:

- The logging systems are enabled
- The logging activation condition for each logging system is selected
- Cyclic logging is selected or deselected
- The channels to log for each logging system are selected
- Logged data can be retrieved and converted
- The logging memory can be erased

See the M1 Tune User Manual for details.

In M1 Build:

- Logging conditions are defined
- Logging default rates are defined
- Channels can be selected for 'Diagnostic Logging'
- An additional diagnostic log rate can be defined to the respective channels
- Tags can be assigned to channels
- If an 'Advanced' security setting is chosen, the security setup for the logging systems is defined.

Logging Conditions

In M1 Tune channels of the built-in data type 'Logging Mode' are used as logging activation conditions. Each logging system has an individual logging activation condition. A logging system will be activated if the assigned channel has a value of 'Run', and logging will be stopped if the channel has a value of 'Stop'.

Such channels must be defined in the Project if logging is intended. The Project developer can define the condition required to provide a value of 'Run', for example when the engine is not stalled.

It is also possible to provide more than one channel with the data type 'Logging Mode', for example to trigger logging of specific logging systems under special circumstances

Logging Rate

In M1 Build, the logging definitions of a channel are part of its properties. Adjust the properties to define the default value for the rate at which to log the value of a channel when selected for logging in M1 Tune. This log rate can be changed in M1 Tune. However, the log rate can only be less or equal to the update rate of a channel (which will be the rate used if 'Default' is selected).

Diagnostic Logging

A channel can be selected for 'Diagnostic Logging' and assigned a specific diagnostic log rate. The selection of Diagnostic Logging channels cannot be changed in M1 Tune. Specified diagnostic log rates can be changed in M1 Tune, but only towards higher values.

Diagnostic Logging channels are considered the highest priority so they will always be logged, at their specified diagnostic rates, when the first logging system is active in the ECU. By designating channels that are likely to assist in troubleshooting, the Project Developer can guarantee the availability of essential information should customer support be required.

As more channels are allocated to Diagnostic Logging, the capacity to accommodate additional logging channels reduces, so consideration should be given to the available logging resources.

If at least one channel is selected for Diagnostic Logging, the activation condition for the Diagnostic Logging must also be defined in M1 Build, and this will become the mandatory logging activation condition for the first logging system in M1 Tune. It is necessary to define a channel that uses the built-in data type 'Logging Mode', as described in Logging Conditions, and assign such a channel as 'Diagnostic Logging Condition'.

The Diagnostic Logging configuration must conform to the limits of the Logging Licence in the target M1 ECU. The Licence requirements for the target M1 ECU can be specified in the Diagnostic Logging configuration. If the target M1 ECU does not meet the minimum Licence requirements, the Package cannot be sent to the M1 ECU.

M1 Build validates that the configured Diagnostic Logging channels conform to the limits of the selected Licence.

Logging System Setup for Advanced Security

If an 'Advanced' security level is defined in the Project, each of the 8 logging systems can be set up with individual permissions for:

- Adjustment in M1 Tune
- Download
- Conversion
- Deletion.

In addition, each channel must be allocated to a specific security group. By allocating a user to certain security groups, a user's ability to log a channel can also be restricted.

See Security for details on 'Advanced' security, and the M1 Build User manual on how to configure the settings.

Glossary

This glossary lists and defines all special terms and acronyms used in this manual.

M1 introduces unique terms and at times uses common terms to represent something specific to the M1 environment.

Term	Definition
Archive	Archives are single files that contain all the information required to correctly install various MoTeC components.
Base Unit	The unit used to store a value of a particular quantity. This is generally the SI unit, with the exception of temperature and angle.
Built-in Class	The lowest level building blocks available in M1 Build. They can be used directly or to build other classes.
Class	A class is a construct that is used to create instances of itself called objects.
Default value	The value a data object is initialised with. In the M1, the default value is 0 for all data objects.
ECU	Electronic Control Unit. Sometimes seen as Engine Control Unit.
Firmware	The firmware is the machine code generated by the M1 Build Project compilation.
Function	A class containing code.
High Level class	A High level class is a construct built up of other classes.
M1 Tune	The application used to interact with the M1 ECU. Tune opens Package files, connects to the ECU and retrieves and sends Packages. Tune shares many features with i2.
M1PL	The M1 Programming Language.
Method	A method is a function associated with an object.
Object	An instance of a class once it is part of a Project.
Package	A Package represents the complete state of an M1 ECU. Packages can be sent to and retrieved from an M1 ECU.
Project	A Project is the complete source-level definition of an M1 ECU application.
Quantity	A physical property that can be quantified by measurement. For example, length or mass.
Scheduled function	A class containing code that is associated with an event.
SDK	The definitions of all system functions and input/output methods available in Build for use in a Project.
Separation of concerns	The principal of structuring a program in distinct sections that address separate concerns.
System	The M1 System defines the operating environment for the Package. It consists of a kernel, drivers for the input/output hardware, file systems and management tasks.